AMRF NETWORK COMMUNICATIONS

NBSIR 88-3816

June 30, 1988

By:
S. Rybczynski
E. J. Barkmeyer
E. K. Wallace
M. L. Strawbridge
D. E. Libes
C. V. Young

U.S. DEPARTMENT OF COMMERCE National Bureau of Standards Gaithersburg, Maryland

AMRF Network Communications

Siegfried (Fred) Rybczynski
Edward J. Barkmeyer
Evan K. Wallace
Michael L. Strawbridge
Don E. Libes
Carol V. Young

June 30, 1988

AMRF Network Communications

Table of Contents

List of Figures

List of Tables

I. <u>INTRODUCTION TO THE MANUAL</u>

1. PURPOSE OF THIS DOCUMENT

This document describes the National Bureau of Standards (NBS) Automated Manufacturing Research Facility (AMRF) factory network architecture as implemented during 1985 thru 1987. Although changes in equipment have been made since 1986, the network architecture description remains accurate.

Further changes effecting the network architecture, topology, and operating procedures are expected as computer integrated manufacturing continues to evolve in the AMRF.

2. ORGANIZATION OF THIS DOCUMENT

This document is logically organized so that the interested reader can receive the appropriate level of information without reading more detail than is necessary. As shown by the table of contents, this document begins with a high level overview of the communications system and then describes the common memory and network architectures in great detail.

Since the Programmer Reference Section and the Operator Reference Section are relatively short, they have been incorporated into this single document.

Specific terms are used throughout this documentation to refer to systems and components of the AMRF. These terms are defined in the glossary located near the back of this document. All readers are encouraged to acquaint themselves with the glossary contents.

3. INTENDED AUDIENCE

3.1. <u>Casual Reader</u>

The casual reader is directed to Section II, System Overview.

3.2. <u>System Implementor</u>

The system implementor should read all sections of this manual.

3.3. <u>Network Programmer</u>

The network programmer should read all sections of this manual.

3.4. <u>System Operator</u>

The system operator needs only to read Section VI.

II. SYSTEM OVERVIEW

1. GENERAL OVERVIEW OF AMRF COMMUNICATIONS

The old idea of the automated factory as a group of machines controlled by one huge central computer lacks flexibility. In the "factory of the future", computer processes such as control programs will run on many different computers, of all sizes and models, and possibly located in different buildings.

Such a "distributed" system requires a method of transferring information which is fast, accurate, reliable, and independent of the actual physical location of the machines. The transfer of information must also be done without interferring with or adversely affecting real-time processes on the receiving machine. This can be achieved either by direct originator-to-recipient message passing or by using a common (shared) memory.

The AMRF implementation is based on a global common memory. Information (data) is deposited into the common memory area by one process and read from the same area by one or more other processes. The writer and reader (producer and consumer) processes can be tightly coupled (i.e., they share the same bus), or they can be loosely coupled (i.e., the accesses are supported by some network pathway).

The AMRF common memory uses the concept of computer "mailboxes", areas of common memory on various computers to which all of the application processes have access. This access is through the network communications system, and is subject to strict rules of protocol [2]. Application processes can leave "messages" for each other and stop to read their own "mail" at opportune times without interrupting each other. The common memory mailbox implementation permits data to be used by more than one process without explicit action by the originator to deliver it to all users. Common memory is particularly effective in equipment level systems which must perform real-time data acquisition and processing.

Currently, the AMRF communications network uses an Applitek broadband token bus and a combination of other computer communications protocols, including RS232 and Ethernet systems. Work is underway to upgrade the AMRF network to one based on the principles of the Manufacturing Automation Protocol (MAP).

2. THE AMRF COMMON MEMORY CONCEPT

2.1. General Description

One of the first designs of common memory was developed to support real-time data reduction and robot control in a multiprocessor configuration using a physically common memory

[1]. Each of the processors (a single board computer) is connected to a special memory area that is usually on a separate board. This area maps into the address space of each processor, allowing it to read and write to this "common memory area" and thereby communicate with the other processors in the configuration (Figure II-1).

The premise is that a multilevel control system could be designed in such a way that each level could execute independently on a separate processor of the multiprocessor configuration. The inputs to the levels - command, status, and feedback data - could be read from one or more common memory buffers. A designated processor computes the function of that level and the corresponding outputs could be written to a second set of common memory buffers and thereby made immediately available to the other processors (levels).

The advantage of common memory is that, as new processes are added that need information already present, extant processes do not have to be modified to deliver that information.

For example, a process was added that displays the robot's actions on a graphics monitor. The display process was added without modification to any other part of the system, since it uses the joint angles which are stored in known common memory locations. Recently, a "safety" process was added to guarantee that the robot never departs its working envelope. The safety process also obtains its information from common memory.

In 1981, work began on an automated factory [11]. This extended the common memory concept of the robot control system in many ways. A major extension is that processes which have to communicate are often in separate backplanes and use different operating systems. A single physical common memory is no longer possible or practical, so each computer system has its own local common memory. These common memories are connected through locally-developed network services that are transparent to the user process, establishing a global logical common memory (Figure II-2).

Common memory provides a consistent communications methodology for this diverse collection of computers and operating systems.

2.2 AMRF Implementation of Common Memory

Common memory has been implemented on different systems in the AMRF using a true hardware shared memory, using message passing to a memory-manager process, and using a process which copies the information between the local memory of each control process and a background common memory.

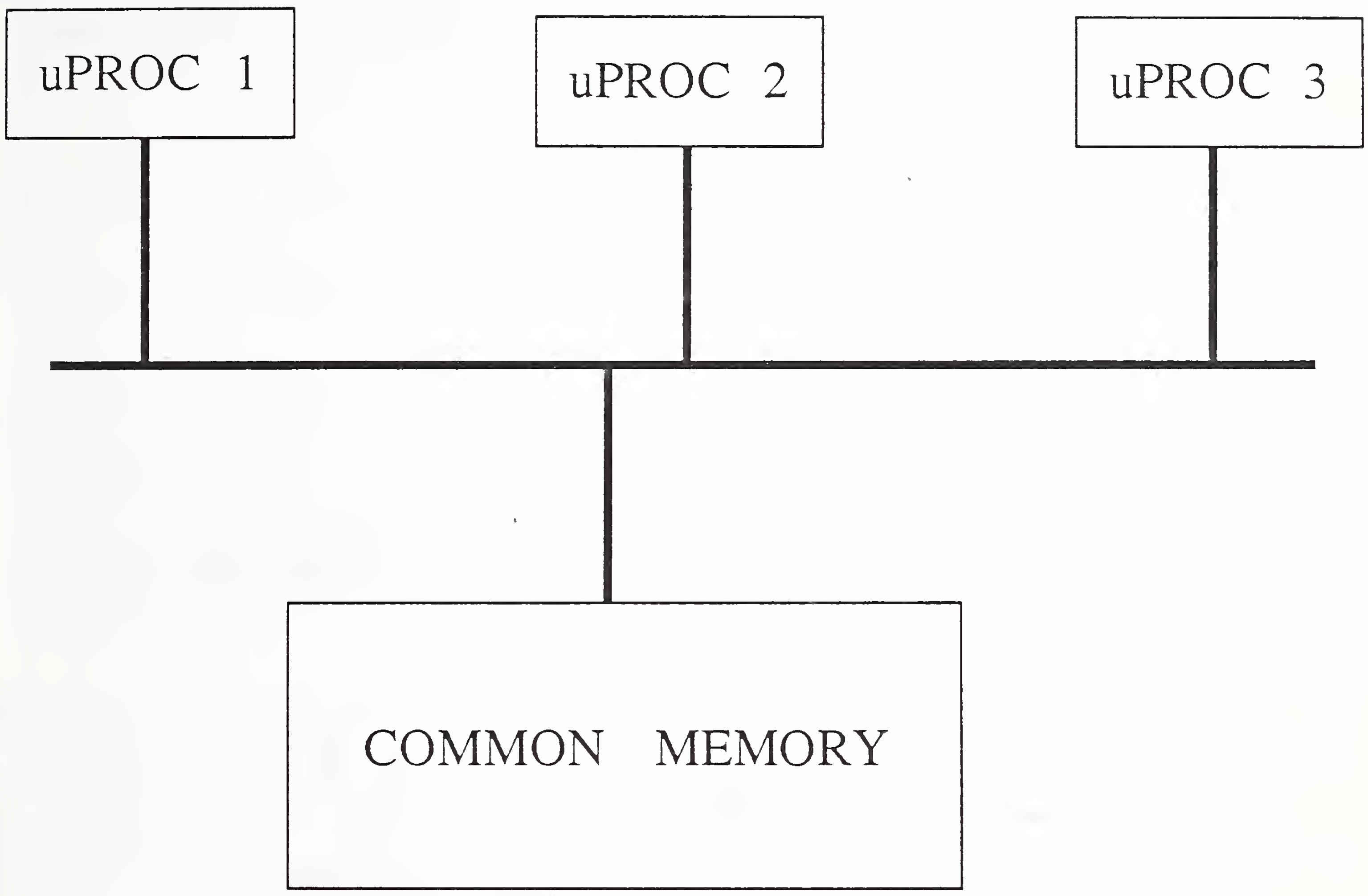

Figure II-1. Multiple-Processor Structure with a Physical Common Memory.

Several microcomputers are connected through a common bus structure to a common memory area that maps into their address space.

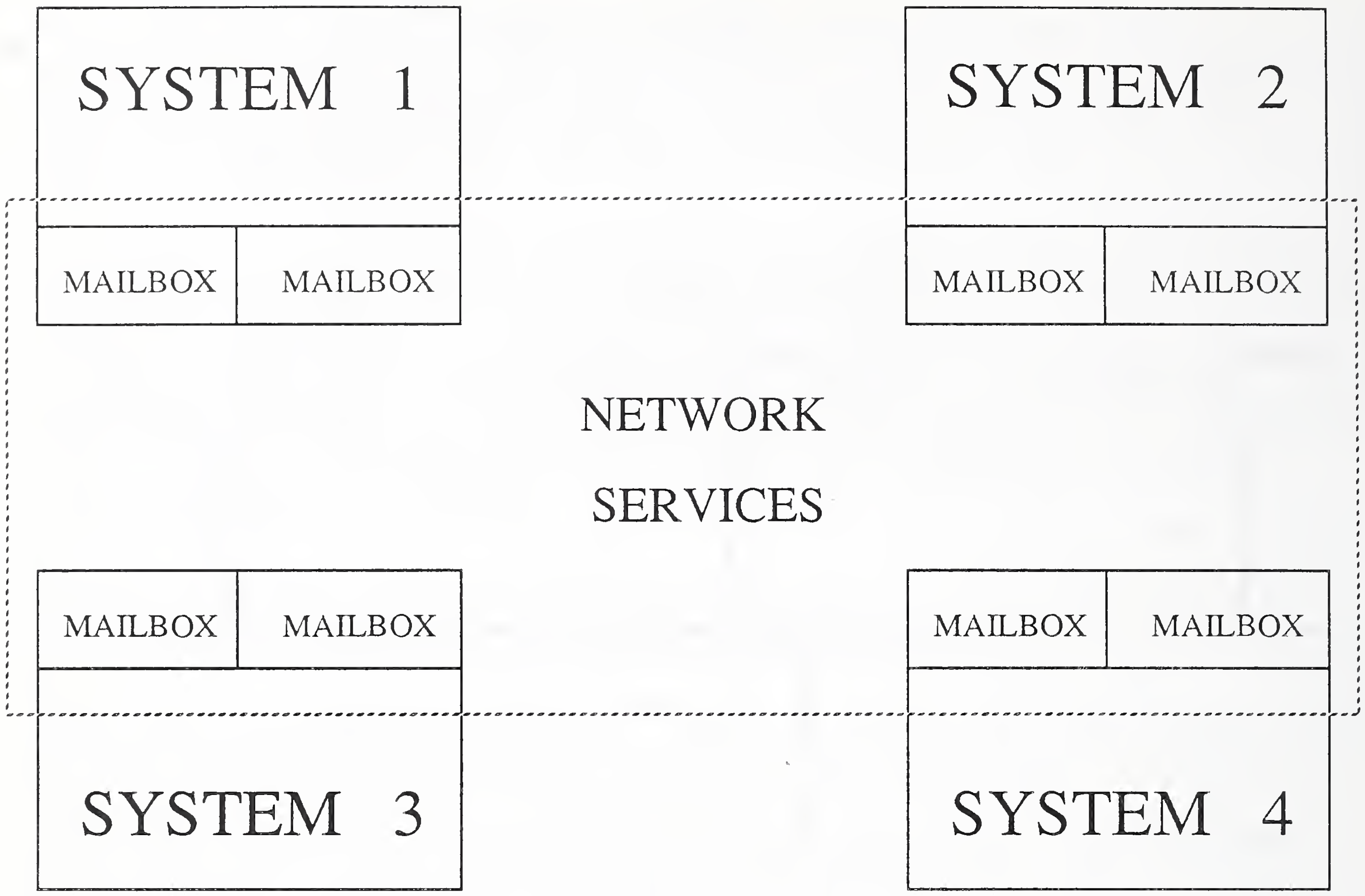

Figure II-2. Global Common Memory Environment

The mailboxes that comprise the local common memory environments in the individual computer systems are connected via network services to create a global common memory.

Common memory access is limited to a predefined set of mailboxes, each containing a single logical record (a "mailgram") which is regularly updated by a direct replacement (rewrite). The originating process rewrites the information unit in the designated mailbox whenever it wants to.

When the retrieving process resides on the same computer system and has direct access to the same memory address area, it can "read" the information unit simply by fetching the mailgram from the common memory space. In the current AMRF topology, this is true of the VAX-based processes.

In Figure II-3, the arrows show the flow of command and status information: application process 1 deposits data into common memory mailbox A, which is read by both application processes 2 and 3. Process 2 generates a message that is deposited into mailbox B for process 3 to read, based on the data in mailbox A.

In other cases, the retrieving process may reside in the same computer chassis (i.e., a Multibus system) but have no direct access to the originator's memory. A transporting process resident on each of the processor boards must copy the mailgram in the originator's mailbox to a separate mailbox designated to receive that information in the local memory area that maps into the address space of both the information generator and the information retriever. This is the case with the component control processes of the NBS Robot Control System [3].

When the retriever resides on physically separate computers, a Network Interface Process (NIP) resident on the originator's node uses the local shared memory protocol to read the originator's mailgram and transmits a copy of the mailgram over the AMRF network to the NIP on the retriever's system; the receiving NIP then stores the mailgram into the appropriate mailbox on that system, where it can be read by the retriever using the local protocol there.

Figure II-4 displays the distributed common memory. As in Figure II-3, information is passed between three processes. However, because these three processes are located on three different (remote) processors, the NIP passes the information through the network and into and out of the appropriate common memory mailboxes. The arrows show the flow of command and status information.

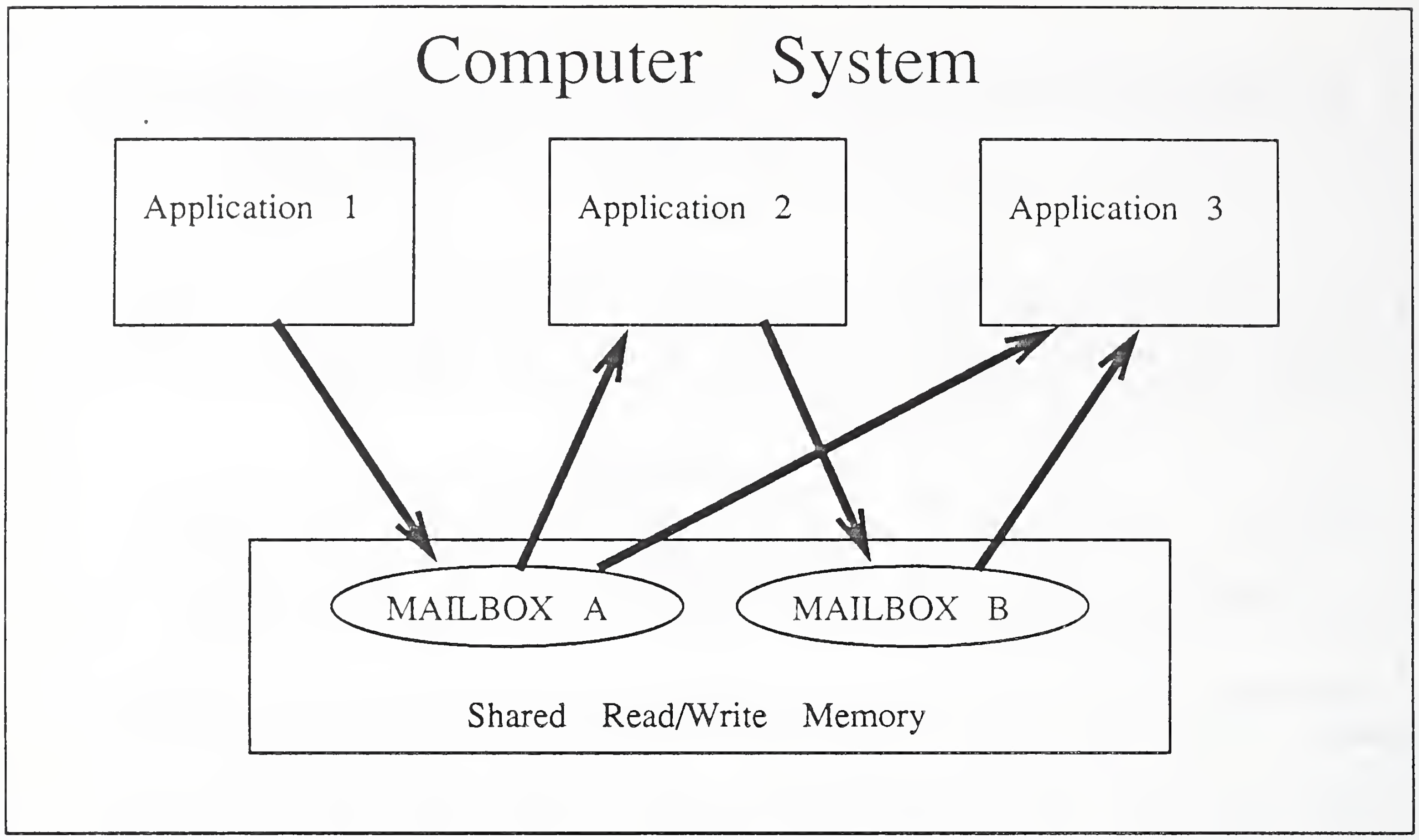

Figure II-3. Local Common Memory.

The arrows show the flow of command and status information: Application 1 deposits data into common memory Mailbox A. This data is read by both Application 2 and Application 3. Application 2 uses this data to generate additional data for Application 3.

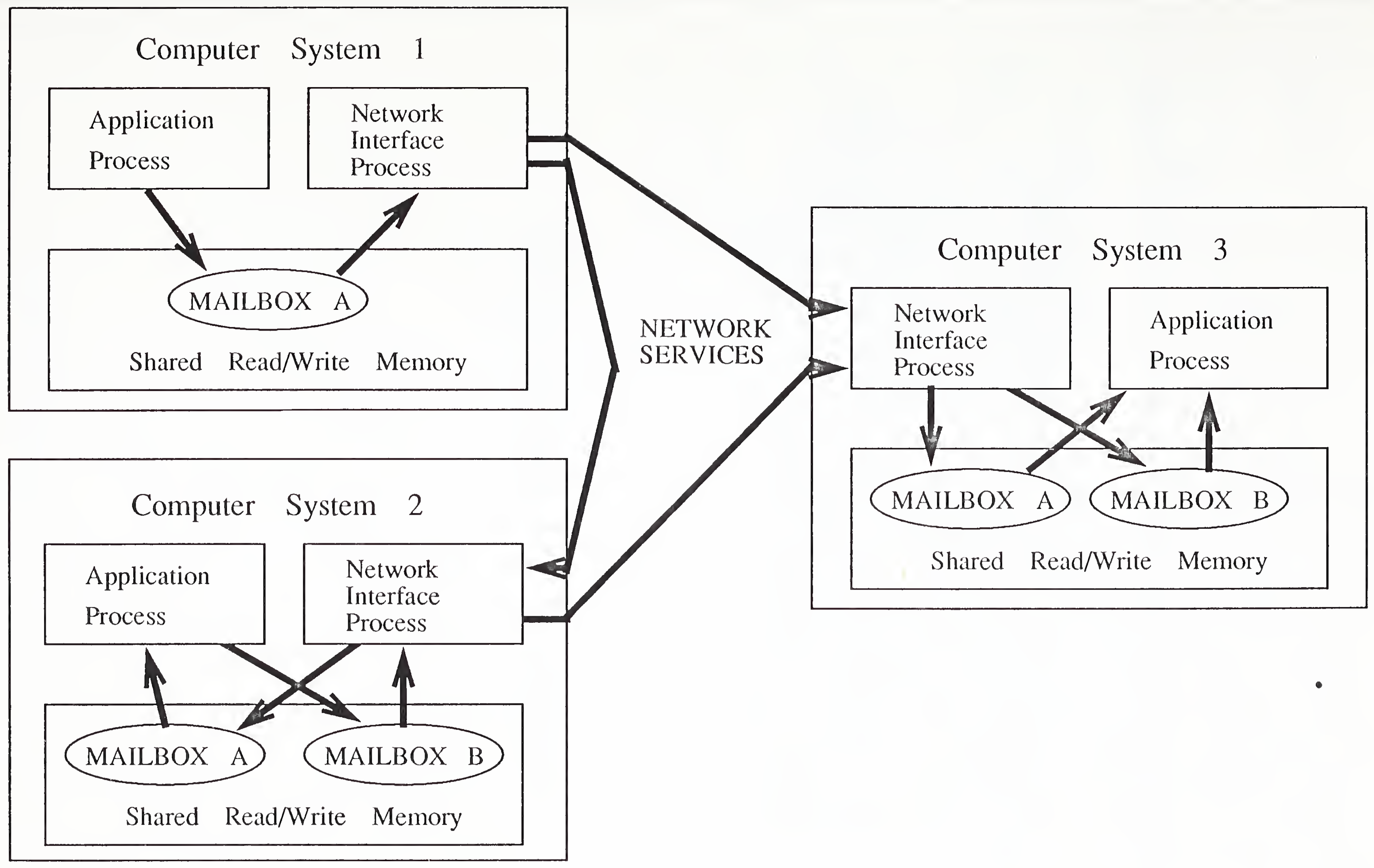

Figure II-4. The Distributed Common Memory.

As in Figure II-3, information is passed between three processes. However, because these three processes are located on three different (remote) processors, the NIP passes the information through the network and into and out of the appropriate common memory mailboxes. The arrows show the flow of command and status information.

The common memory implementation on the Sun Microsystems computers (hereafter refered to as "Sun"), under 4.2 BSD Unix, has a problem: there is no direct common memory support facility. This is circumvented by providing a server process that stores mailgrams in its own memory space, which is private to it. Communication occurs between user processes on the sun (or any other sun on the same network) and the server process using standard 4.2 BSD interprocess communications [12]. The NIP is simply another process that exchanges mailgrams with the common memory server (Figure II-5).

In all cases, the control process view of the communication is that it stores into or fetches from a shared memory area, according to some protocol common to all processes on that system. Thus the originating process does not have to know where the retrieving processes are, or even which ones they are, as long as it abides by the local protocol; and the retrieving process does not have to know where or how the information originated. To both processes, only the structure and function of the information are significant. This mechanism encourages the development of standard functional information groups, to be created and consumed by control and sensory processes, without regard for the mechanics of interprocess communication.

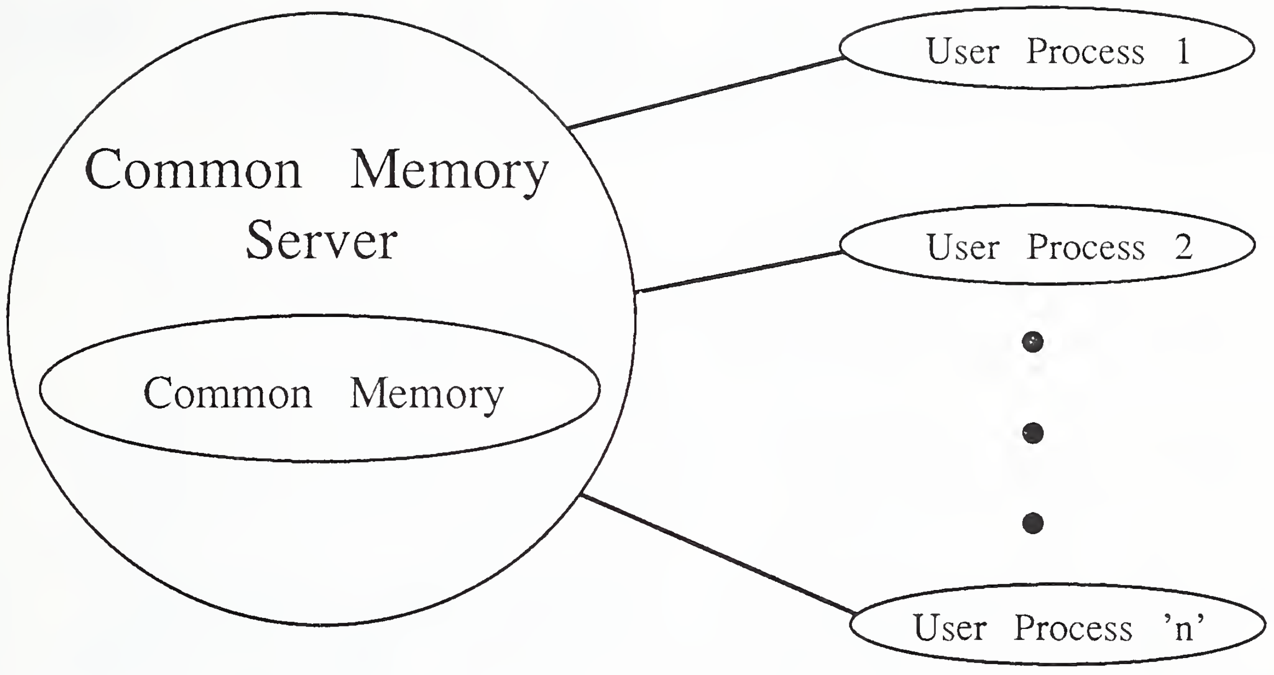

Figure II-5. Common Memory Server For 4.2 BSD Unix

3. THE AMRF NETWORK

3.1 Network Topology - 1986

Figure II-6 shows the topology of the 1986 AMRF network.

The network is primarily comprised of point-to-point connections using serial, RS232 connections. Recent additions have provided additional network pathways using Ethernet (TCP/IP) to accommodate greater traffic loads while simultaneously providing enhanced speed.

Two subnetworks are identified:

(1) at the Inspection Workstation (serial RS232), and
(2) at the Horizontal Workstation (Ethernet).

The purpose of the subnetworks is to isolate large volumes of local traffic from the primary network pathways. This has the advantage of enhancing overall communications performance.

CELL, MHS, PPL, VWS, and CDWS, do not currently run a resident version of common memory and the network interface process. They are all interfaced to the global AMRF common memory through a front end common memory server system. The interface uses TCP to pass messages between the common memory server and its clients. Except for CELL and MHS, the clients communicate directly with the front end common memory.

CELL and MHS run on personal computers (PCs) and utilize a secondary communications system (a locally-developed program, see Section V.2.6.) to exchange mailgrams between the PC and a server process on the common memory front end machine. This server process then performs TCP communications with the common memory server process.

We expect the AMRF network to evolve into one predominantly based on the MAP architecture and composed of one backbone network with several subnetworks. The pace of this evolution is dependent on the availability of new commercial products and the continuing evolution of the MAP standards. Other network architectures, such as the Department of Defense Internet Architecture (TCP/IP and associated protocols), will undoubtedly persist within the AMRF for many more years.

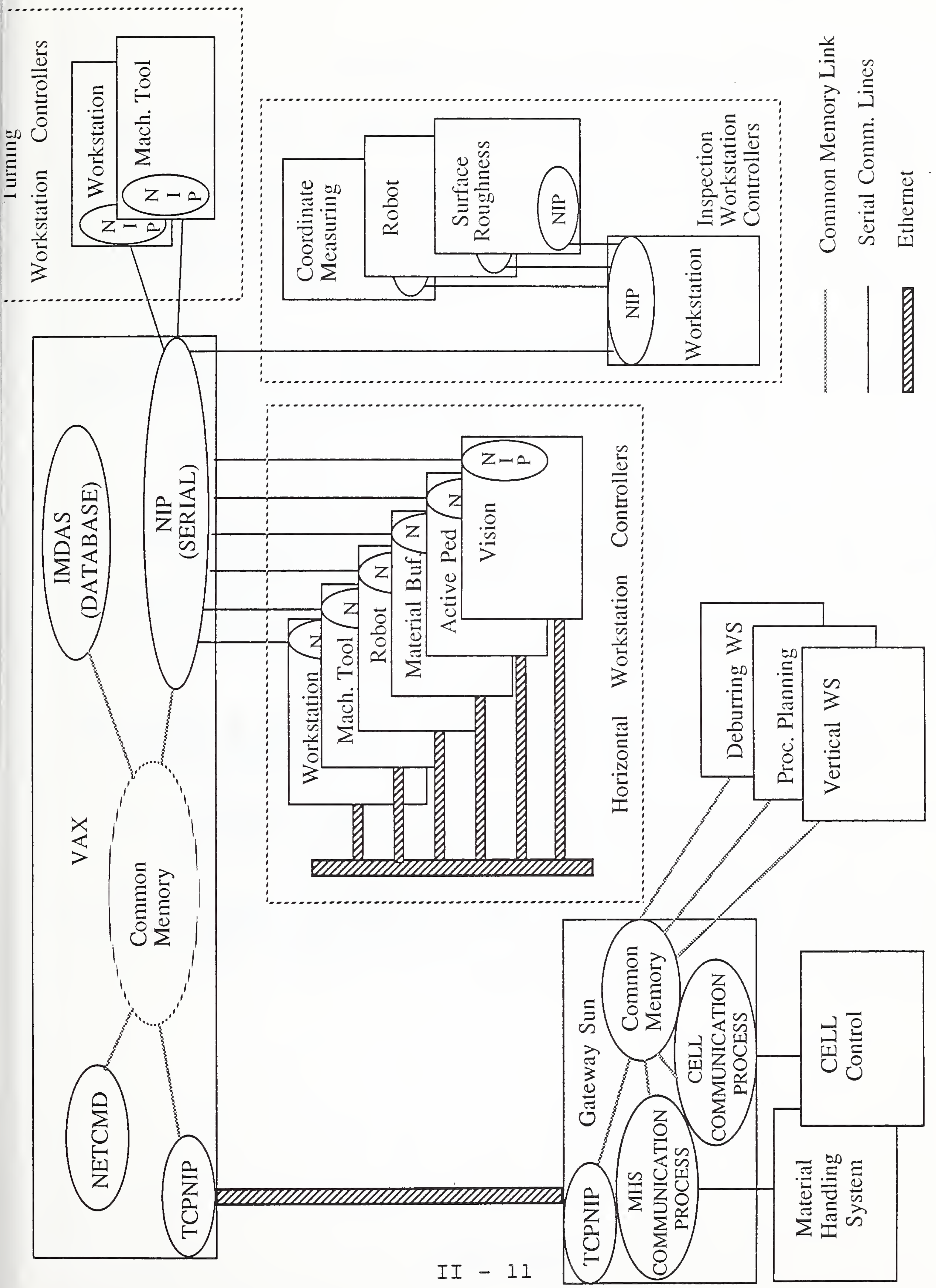

Figure II-6. The Topology of the 1986 AMRF Network

3.2 Mailgram Delivery Over the Network

The actual mail delivery processes - the local transport processes and the network interface processes - are simple table-driven machines.

During its execution cycle, a local mail delivery process:

(1) examines the mailbox that contains commands addressed to the delivery process. If a command is found, the process modifies its mail delivery table as directed.

(2) makes one pass through the delivery table, copying each eligible mailgram from source to destination. The usual practice is to copy from the originator's mailbox into the "shared memory" and from the shared memory to the retrievers' mailboxes in order to avoid coupling semaphores.

Figure II-7 depicts a network interface process (NIP). NIP tables are similar to local transport tables, except that in each entry, one of the source and destination mailbox identifiers is replaced by a network locator. The NIP cycle consists of

(1) examining its command mailbox and modifying its delivery tables as directed,

(2) processing all outbound table entries,

(3) copying the contents of any mailbox which has changed into a network packet and routing it to the specified network location, and finally,

(4) looking up the table entry for each received packet and depositing the mailgram in the proper local mailbox.

In the current AMRF implementation, the commands that tell the mail delivery routines to build their tables are issued through a network manager process. The inputs come directly from a human operator, who effectively manages the shared memory data directories (the mailboxes) on paper.

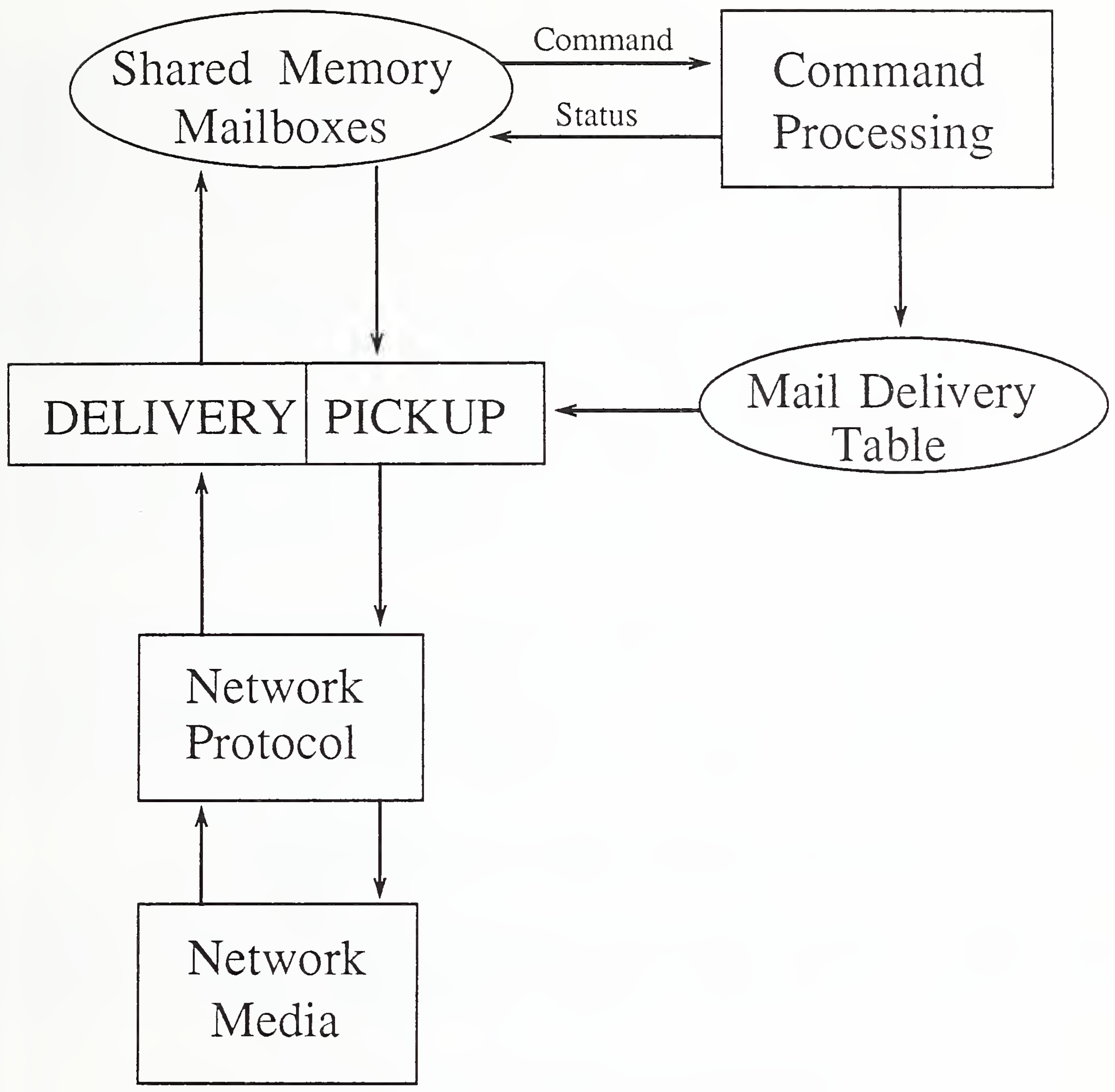

Figure II-7. The Anatomy of the Network Interface Process (NIP)

III. COMMON MEMORY ARCHITECTURE DESCRIPTION

1. MAILBOXES AND MAILGRAMS

All interprocess communication is accomplished through a mechanism called "mailboxes". Mailboxes are logical storage areas where messages (called "mailgrams") are placed by the sender process and picked up by receiver processes. From the point of view of the sender and receiver processes, the location of the correspondents does not affect their communication.

These mailboxes reside in a special area of memory, designated local common memory. Local common memories are combined over the AMRF network to form a global common memory (Figure III-1).

The user interface to the local common memory area may be implicit or explicit (see Section III.3.1, below). Implementation of an explicit common memory interface implies the presence of a common memory manager function to dynamically create and destroy mailboxes and coordinate mailbox access.

The mailgram transfers across computer systems are accomplished by the communications systems in a fashion totally invisible to the sender and receiver processes. Moreover, the communications system operates asynchronously. That is, it does not require the sender to wait for the receiver to get the mailgram, or the receiver to pick it up immediately when it arrives.

1.1 Coordinating Common Memory Access

Common memory environments can be susceptible to several problems related to coordinating access to these areas. Each potential problem, however, has one or more solutions. All of these methods are used within the AMRF.

1.1.1. Read While Write is Active

A process may be attempting to read information from a common memory area at the same time as a second process is attempting to update that same area. As a consequence, the reading process may get inconsistent information (e.g., the current value of field A and the former value of field B).

Methods to avoid this are:

(1) use a semaphore for each common memory buffer area (a mechanism that supports single-process access to the buffer). Some processors (HP 9000 Series 200 and MC68000) provide atomic "test and set" operations which can be used as hardware semaphores. Software semaphores, using Dekker's algorithm [10] for example, can be extended to provide mutual exclusion between any number of processes.

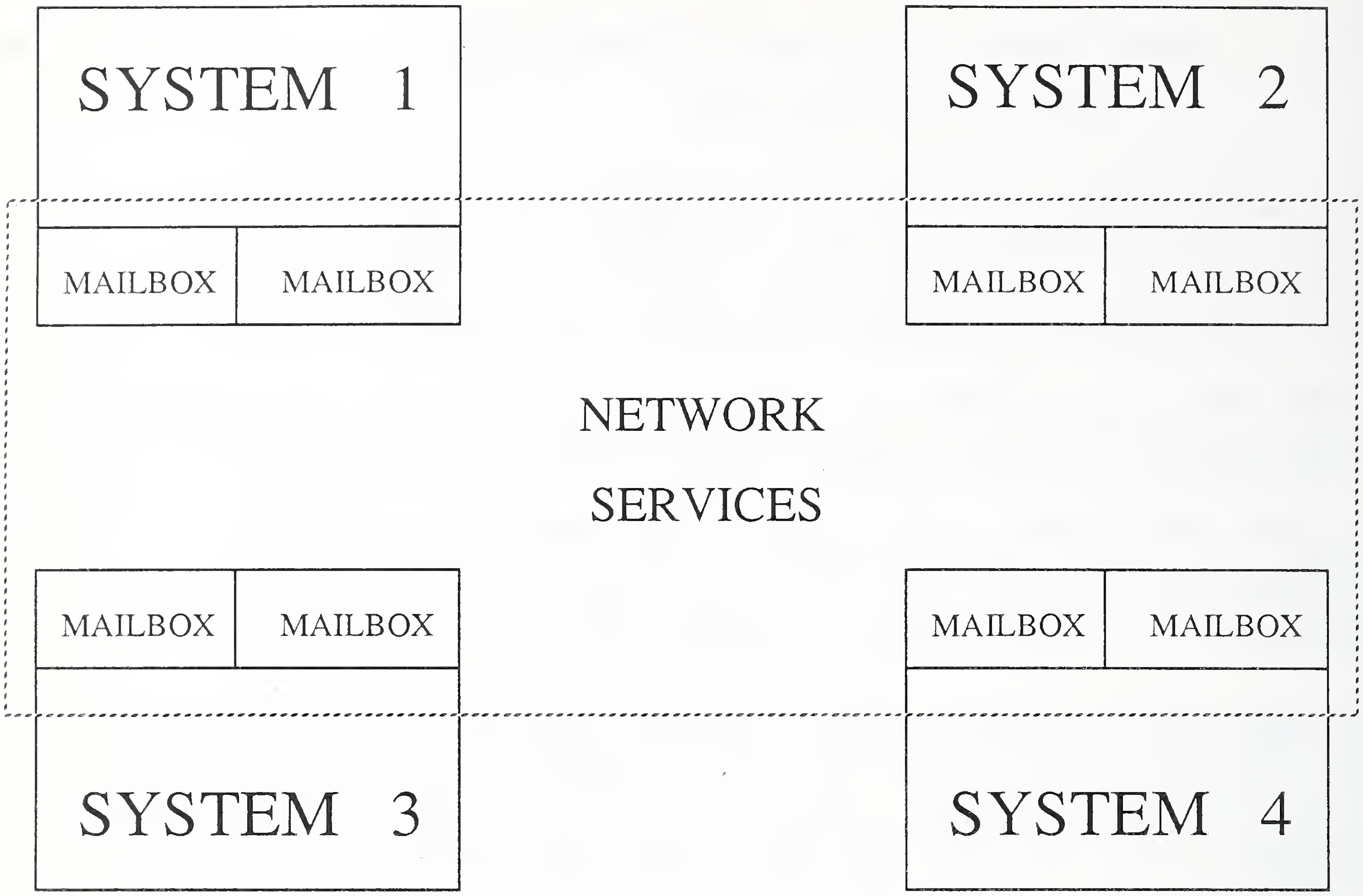

Figure II-2. Global Common Memory Environment

The mailboxes that comprise the local common memory environments in the individual computer systems are connected via network services to create a global common memory.

(2) define a regular, recurring real-time interval and divide it into a write-only period and a read-only period. Any process not prepared to perform a write operation during the write-only period would have to wait for the next write-only period. The same restriction holds for read-only periods.

(3) pass a token among participating processes. The process that has the token can perform any read or write operation it wants. Fixed length or varying length time quantums can be employed. Token passing has an unfortunate drawback: if the process with the token halts (or appears to do so), passing of the token becomes impossible and all access to common memory is barred. In a fixed length time quantum implementation, the token can be reissued by some governing process after the expiration of the time quantum (plus some extra "safety margin"); in a varying length time quantum implementation, the recovery algorithm is much less obvious.

(4) utilize a hardware architecture that does not support interrupt processing. Once a processor has control of the bus (and consequent access to common memory), no other processor can interrupt, thereby assuring that overlapped access does not occur.

1.1.2. Update Frequency Exceeds Read Frequency

A process may update the common memory area more often than a reading process is able to retrieve the information.

This may only be an "application-specific" problem. That is, if the reader process only wants the "current" information (as from a sensor, for example), then the fact that any amount of older information may have been missed is a moot point. However, if it becomes important that the reader process have access to each information set before it gets updated, then some form of "flow-control" must be used.

For example, if the information set in a common memory buffer is uniquely identified (a time stamp or sequence number), then flow control could be implemented by defining a second buffer in the common memory area into which the reader process could echo the unique identifier. When the writer of the original information sees the echoed identifier in this second common memory buffer, it knows that it can proceed with the next update.

1.1.3. Read Frequency Exceeds Update Frequency

A process may read the data in the common memory area more often than a second process is able to update it. This can result in "old" information unintentionally being considered "new" information.

In the case where the information happens to be a command such as "hit nail on head with hammer", an undesirable number of duplicate executions could be performed.

A solution is to identify new information whenever it is placed into the common memory buffer by implementing a flag field within the buffer. This flag field could take the form of a sequence number that gets incremented with each update of the buffer, or a time stamp that identifies when the information was placed into the buffer. In each case, the reader process is looking for a change in the flag field to indicate that buffer contents have been updated.

1.1.4. Multiple Readers Of A Common Mailbox

In the case where a single mailbox is being accessed by multiple readers, if it is important that each of the readers have the opportunity to retrieve the mailgram before it is overwritten, then a more elaborate form of flow control must be implemented.

One solution is to share a single "flow control" mailbox between all the readers. Each reader sets a specific "flag" in the mailbox indicating he has retrieved the message. When all flags have been set, the shared-read mailbox contents can be overwritten. This "solution" immediately introduces another problem: multiple writers to a single mailbox. (See Section III.1.1.5.)

A simpler, more reliable solution is to assign each reader process its own flow control mailbox.

1.1.5. Multiple Writers To A Common Mailbox

Unpredictable results can occur when more than one process is permitted to write into a single common memory buffer:

(1) predicting the sequence in which information is written to the common memory buffer may be impossible,

(2) guaranteeing that all reader clients have seen the contents of the common memory buffer before it is updated may be impossible,

(3) identifying the intended reader client audience for any particular memory buffer update may be impossible.

A simple solution is to stipulate that any common memory buffer is permitted to have only a single process writing data into it, although it can have any number of reader clients.

More complex solutions that support the use of a single common memory buffer by more than one writing process are possible. In general, these solutions require the implementation of enhanced flow control and flag field techniques.

1.2. Mailgram Format

In general, there is no standard format for a mailgram. There are, however, standard information units which must be carried either in the mailgram, or with the mailbox by the common-memory services. These are:

1. Length of the current mailgram in the mailbox,
2. Sequence number or other index of change (see below),
3. Access control flags.

When all of these entities are expressed in the mailbox area itself, the mailbox has the structure shown in Figure III-2.

Some systems implement all of these units in the mailbox area, (e.g., Multibus multiprocessor systems); some implement only the length and sequence units in the mailbox area (e.g., VAX and HP); and some implement only the text in the mailbox area (e.g., SUN).

Read and write locks, when implemented, are considered to be part of the mailbox and not the mailgram. That is, when the mailgram is transported to another network node by the NIP, the lock bytes are not transported.

```
Byte  1      2      3      4      5      6      7      8
  ---------------------------------------------------------
  ¦      ¦      ¦      ¦      ¦      ¦      ¦      ¦      ¦
  ¦  Write Lock ¦  Read Lock  ¦   Sequence  ¦   Length    ¦
  ¦      ¦      ¦      ¦      ¦      ¦      ¦      ¦      ¦
  ---------------------------------------------------------

          9      10     11     12    ...
        ----------------------------------
        ¦      ¦      ¦      ¦      ¦
        ¦ Process-Dependent Text . . .
        ¦      ¦      ¦      ¦      ¦
        ----------------------------------
```

Write Lock	is a semaphore indicating current writer activity. (i.e., if the write lock is ON, then the mailbox is being written, and should not be read)
Read Lock	is a semaphore indicating current reader activity. (i.e., if the read lock is ON, then the mailbox is being read, and should not be updated)
Sequence	is a sequence number attached to the mailgram in the particular mailbox. Every time the text of the mailgram is changed, the sequence number is incremented. The update can be detected by examining only the sequence field.
Length	is the length of the mailgram in bytes.
Text	is the information portion of the mailgram that is defined entirely by the communicating processes.

Figure III-2. Generic Mailbox Structure

1.3. Mailbox and Mailgram Properties

(1) The mailbox must be created (or declared) before mailgrams can be deposited into it. If the network is to deliver copies of the mailgram to other, remote locations, then the mailbox must exist and a network connection must be created between the sender and receiver of the mailgram before any mailgram can be delivered.

(2) Every mailbox has a unique global (AMRF-wide) name. The name is assigned to the mailbox at the time it is created and identifies the mailbox to the entire AMRF. That is, remote systems desiring a copy of the mailbox contents must have a mailbox with the same name available in their local common memory (this is a convention enforced by the human network manager). On some systems, mailbox naming only has local significance and the transfer of information is actually to and from an address (i.e., explicit systems).

(3) Every mailbox contains an initial value assigned by the creator when it is created. In some systems this is a standard value (e.g., all zeros); in other systems, this value defaults to the contents of memory at the time of creation.

(4) Every mailbox contains exactly one mailgram at any given time. A mailgram stays in the mailbox no matter how often it is read, until a new mailgram arrives for that mailbox. The new mailgram replaces the old one on arrival, whether or not the old mailgram has ever been read.

(5) The mailbox writer decides when to replace the mailgram. This may be performed independent of external information, or may be influenced by "flow control" factors. (Section III.1.1)

(6) Only one process is authorized to write into a mailbox at a time. In the case where more than one process may write into a specific mailbox, implicit or explicit "flow control" is implemented to match others. (Section III.1.1)

(7) Two mailboxes must be "connected" before a mailgram will be transferred from one to the other by the network. Global mailboxes are "connected" and "disconnected" by submitting the appropriate command to the network manager. (Section IV.4)

(8) Any number of receiver processes can pick up the current mailgram in a mailbox.

(9) Any receiver process can pick up the same mailgram several times, if the sender does not change it in the interim, or miss several mailgrams if the sender changes it more often than the receiver picks up. When it is important to assure that a particular recipient has read the mailgram before a new one is issued, the sender and receiver must agree to a "flow control" protocol.

The mailbox management mechanism guarantees that a new mailgram will be distinct from its predecessors. However, the mailbox management mechanism does not guarantee that any particular receiver will have picked up a mailgram before it is replaced. If it is necessary to assure that a particular receiver has read the mailgram before it is replaced, the sender and that receiver must agree to a protocol by which the sender refrains from replacing the mailgram until it has an indication that the receiver has read it. (Section III.1.1.)

(10) Every mailbox has a fixed size which is defined when the mailbox is created. There is no AMRF-wide maximum on mailbox size. There may be a maximum mailbox size for individual systems, caused by hardware or software limitations. Any given mailbox must be large enough to contain the largest mailgram agreed upon between the sender and receiver(s).

(11) Mailgrams can be of variable length; each mailgram contains information on how long it is. A mailgram may never be longer than the mailbox in which it is placed. If necessary, the value is truncated by the common-memory service routines.

2. MAILBOX INTERFACE IMPLEMENTATIONS

The internal workings of the mailbox operation are unique to each system, while the transmission of mailgrams from system to system uses a common network and common protocols. From the user point of view, however, the method of communication between processes is independent of the location of the correspondent. The user program must use the common memory interface appropriate to the system on which his process resides, and that interface will represent one of two standard methods, implicit or explicit.

When the communication is between processes on the same system, the receiver normally reads the same physical memory area that the sender wrote. When the communication is between processes on different systems, the networking software copies each new mailgram from the sender mailbox to an intermediate mailbox representing the sender mailbox on the receiver's system, and the receiver then reads from that mailbox.

2.1. Implicit Systems

When the implicit method is used, all processes access the same physical memory area for a specific mailbox. A process fills its input buffers from the incoming mailboxes before each processing cycle and empties its output buffers into the outgoing mailboxes at the end of each processing cycle. It is possible in an implicit transfer system, therefore, for processes with fixed intercommunication requirements to be totally ignorant of the interprocess communication discipline, except for the format of the mailgrams. This method is used by the VAX and Multibus-based systems.

2.2. Explicit Systems

When the explicit method is used, each process associates an internal "logical unit" number with a common memory mailbox. This logical unit number is supplied to the process when it creates the mailbox through the respective common memory service. The process then references the logical unit number when it performs PUT or GET operations in order to exchange mailgrams with the common memory.

2.2.1. Special Case: Sun Implementations

The common memory implementation on the Sun Microsystems computers (hereafter referred to as "Sun"), under 4.2 BSD Unix, has a problem: there is no direct common memory support facility. This is circumvented by providing a server process that stores mailgrams in its own memory space, which is private to it. The server runs as a user-level process which needs no special privileges, and can be run from an unrelated user-id [13].

Communication occurs between user processes and the server process using standard 4.2 BSD interprocess communications using TCP/IP rather than UDP [12]. (UDP was rejected because it forced confrontation with communications unreliability and mailgram fragmentation: datagrams were limited to 1K bytes, which was smaller than the typical message.) Access to mailboxes is through valid requests to the server process. The messages are sent asynchronously by the user process, but arrive synchronously: the server process is never interrupted. Typically, the server is waiting for service requests and responds to them immediately.

The network interface process is simply another process that exchanges mailgrams with the common memory server (Figure III-3).

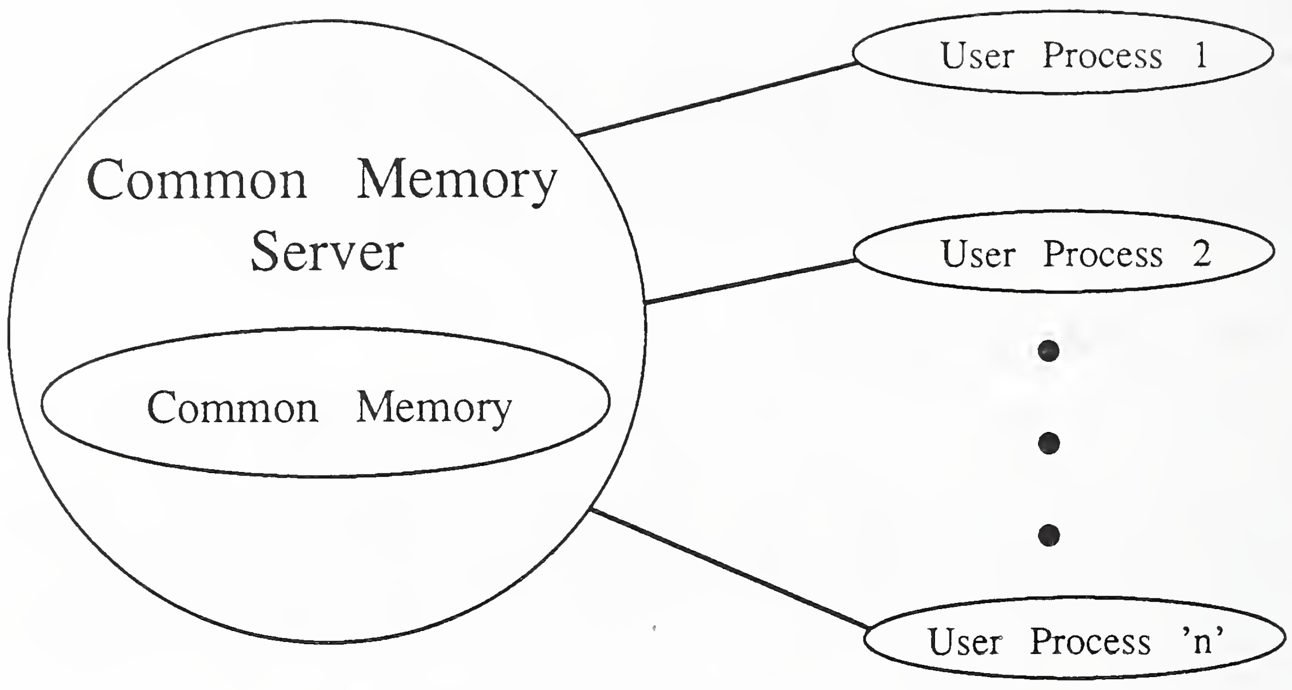

Figure III-3. Common Memory Server For 4.2 BSD Unix

2.3. Conversion of Implicit to Explicit Systems and Explicit to Implicit Systems

Application processes designed for an implicit transfer environment can be moved to an explicit transfer environment by inserting the necessary GETs in a preprocessing routine, the necessary PUTs in a postprocessing routine, and DECLARES (if they are not already used).

Programs designed for an explicit transfer environment can be moved to an implicit transfer environment only if the programs satisfy the architectural requirements of "process-cycle" implementation, that is, the programs must loop through an activity cycle of three parts:

(1) GET all input mailgrams,

(2) perform analysis and determine outputs,

(3) PUT all output mailgrams.

In this case, the GET-input and PUT-output sections of the code can be deleted and the mailbox create commands modified to identify the target buffers.

3. MAILBOX MANAGEMENT

A wide variety of common memory access interfaces have been developed for implementation on the diverse computer systems within the AMRF. The majority of them are used to provide enhanced common memory access and depend on the availability of an intelligent common memory manager, one that performs more than just simple mailbox updates (e.g., notification of mailgram arrival). Due to memory space and system architecture constraints, not all are appropriate for every computer system.

The following subsections describe the functions that are common to all computer systems: DECLARE, UNDECLARE, GET, PUT. An additional one, SYNC, is described because of its importance in a number of computer systems (VAX and Suns).

3.1. Create (DECLARE) a Mailbox

All common memory mailboxes already exist in implicit systems, since they are referenced by address, and the address is not permitted to change without prior notification to all client processes. In this case, a DECLARE connects the user process to the already existing mailbox.

For explicit systems, mailboxes are created or attached by a DECLARE operation. That is, if they don't already exist in the common memory area, they will be created; if they already exist, they are attached. Indications of success or failure are returned from the common memory manager. An internal "logical unit" number is associated with each common memory mailbox, and is used for all subsequent common memory interactions referencing that mailbox.

Some implementations of common memory allow for an access qualifier to accompany the mailbox declaration. This access qualifier can have the following values:

(1) READER, if the connecting process only intends to retrieve mailgrams from the mailbox. This can be further qualified to indicate that the reader process wants "WAKEUP service" so they can "sleep" until the mailbox contents have changed. Nothing happens if the user process is already awake (active) when a WAKEUP arrives.

(2) WRITER, if the connecting process intends to write into the mailbox. This can be further qualified to indicate EXCLUSIVE or NONEXCLUSIVE writer access.

3.2. Discontinue (UNDECLARE) a Mailbox

In explicit systems, mailboxes are released from common memory by an UNDECLARE operation, by which the user process breaks its logical connection to the mailbox. The common memory manager maintains a list of processes that are connected to a specific mailbox, and only releases a mailbox after the last process has UNDELCAREd it. When a mailbox has been UNDECLAREd by all its clients, the common memory manager will return the allocated space to the "free space" pool so it can be used again for other, new mailbox declarations.

In the current software version, the common memory manager in implicit systems does not "resorb" the undeclared mailbox area.

In addition to user processes, the network interface process (NIP) connects to and disconnects from a common memory mailbox when instructed to do so by the network manager. The fact that a user process makes or breaks a common memory mailbox connection has no impact on the NIP connections.

A process should arrange to UNDECLARE every mailbox that it explicitly DECLAREs.

3.3. GET and PUT Operations

There is no common memory management requirement to transfer data between a common memory mailbox and a user buffer in an implicit transfer system, since the user performs this transfer directly. However, for explicit transfer systems the user must issue GET and PUT calls at appropriate points in the processing cycle.

GET results in the retrieval of a mailgram from a common memory mailbox. PUT results in the output of a mailgram to a common memory mailbox.

Through experience with the different common memory implementations, the GET interface to common memory has evolved into GET and GET_NEW, where both function as previously described, but GET_NEW returns an additional Boolean value indicating TRUE if the mailgram has changed since the last time the mailbox was read, and FALSE otherwise. This enables the user process to expedite mailgram processing by providing immediate identification of new mailgrams.

3.4. SYNChronizing With Common Memory

Some explicit common memory systems attempt to provide the user process with copies of all updates of a particular mailbox to which it has DECLAREd a READ connection. (e.g., the Sun implementation.)

Each mailgram that arrives for the declared mailbox is queued until the user process requests a GET. A GET simply returns the next mailgram in the specified mailbox's queue. Hence, it is possible that the user process variable values match a past snapshot of the real common memory rather than the current value.

After returning from a common memory SYNC call, the process buffers (local copies of the mailgrams) are guaranteed to match those of the real common memory. SYNC reads all queued mailgram updates and applies them to the appropriate process buffer.

Several styles of synchronization are possible:

(1) WAIT (sleep), after applying all queued updates, until a declared (with WAKEUP) mailbox's contents changes before returning. Use of WAIT implies that at least one mailbox has been declared WAKEUP or the process will wait forever.

(2) NO_WAIT returns immediately after synchronizing.

(3) WAIT_AT_MOST_ONCE waits for any declared mailbox to change and then returns immediately; the mailbox does not have to be declared WAKEUP.

(4) WAIT_FOR_ALL waits until all declared mailboxes have received at least one update before returning. Some contents may have changed more than once.

(5) WAIT_FOR_READ not only applies all queued mailgram updates, but also forces a new read of all declared mailboxes to transfer mailgrams from common memory to the process buffer.

4. GLOBAL MAILBOX CONNECTIONS

In the 1986 AMRF network, the user process does not have the ability to generate network connections. That is, connections between common memory mailboxes on computers connected by the network are established through operator commands issued through the network manager (NETCMD, Section IV.4).

The only mailbox for which a network connection is automatically requested by the NIP is the NIP's own NIP_CMD input mailbox. It is through this mailbox that the NIP receives instructions from the network manager to establish or remove additional mailbox connections, and the logical network (mailbox) links are established.

IV. NETWORK ARCHITECTURE DESCRIPTION

1. NETWORK MODEL

At the time of inception of the AMRF (1981), there were no national standards by which machines and controllers of various manufacturers could be expected to intercommunicate. There was, however, the ISO 7498 Open Systems Interconnection (OSI) model, and it is upon this model that the AMRF network model and its implementations are based [5].

The "OSI model" specifies the separation of concerns and responsibilities into seven "layers" of software and hardware, loosely defined as follows:

1) Physical: provides direct mechanical and electrical connection between computer systems or network nodes.

2) Data Link Layer: provides for the reliable transfer of information over the physical link.

3) Network Layer: provides for the establishment of host-to-host connections and the end-to-end routing of individual messages when multiple networks or intermediaries are involved.

4) Transport Layer: provides for reliable host-to-host transfer of information over the total network.

5) Session Layer: provides for the differentiation of distinct conversations (e.g. for different users) between the same two hosts, and for management of the individual conversations.

6) Presentation Layer: provides for conversion of information units between local and interchange representations.

7) Application Layer: many different "service elements", each providing a different class of data interchange or data management service.

The OSI model anticipates that for each of the above layers, a standard, or possibly a choice from a collection of standards, will be specified in any given network installation. Since 1984, there has been significant ISO activity in the development of such standards, and there is now a major manufacturing industry initiative to specify such standards across American industry, called the "Manufacturing Automation Protocols" or MAP standard.

Important in the AMRF Network model from the beginning, and only recently added to the MAP concept, are the following two notions:

1) A factory floor network is fundamentally a multi-network, not a single network. That is, a manufacturing network is really a collection of small networks for various specialized activities linked together. Ideally, this linking is performed in such a way that the individual controllers do not have to be aware of the interconnections. This is so in any environment in which real-time activities are networked: one must be able to constrain the traffic on the network carrying the time-critical communications without seriously limiting the overall factory communications capability.

2) The manufacturing engineering and administration systems must be connectable to various factory floor controllers in a substantially automated manufacturing operation. This is so because the scheduling of factory floor operations depends significantly on customer orders, source materials inventory, scheduled maintenance activities and the operating state of various manufacturing components, while the implementation of those operations depends directly on the engineering information, such as process plans and controller programs.

2. PROTOCOL SPECIFICATIONS

Beginning in 1982, the AMRF sought to select then-emerging standards for the protocols in the layers of the AMRF network. Some of the AMRF standards choices fortuitously coincided with the MAP choices; others, largely owing to the availability of limited choices in 1982-4, did not. The currently operating AMRF network is largely a collection of interim protocols which are intended to be gradually supplanted by commercially available nonproprietary network protocols, as individual component systems were upgraded or fit into the whole complex.

One of the features of the AMRF network software, which was made mandatory by the establishment of interim protocols with gradual phaseout rather than abrupt replacement, was the implementation of the OSI model as intended. That is, the software implementing a selected standard for one layer allows for the substitution of protocols in the layer below it, and may in fact be required to support use of different protocols in the next lower layer for different target hosts. (This is not a feature of most of the "MAP-compatible" software products, which makes phased-in conversion extremely difficult.)

2.1. Link and Physical Layers

We pair the Link and Physical layers because in many cases they are paired in the available products, and because the choice of data link protocol often depends on the characteristics of the underlying physical protocol.

This is the first and most significant area in which the AMRF network requires the support of alternative protocols, which are listed below as "interim" and "final" alternatives.

2.1.1 Interim Alternative: Serial Asynchronous Link

The physical protocol is EIA RS232C [17] full-duplex, asynchronous, point-to-point modemless connection at 9600-baud using 3 pins: Transmit Data (TxD, pin 3), Receive Data (RxD, pin 2) and Signal Reference Ground (GND, pin 7). (The cabling is 8-wire, providing for presence (DTR, RLSD) and access-control (CTS, RTS) signals as well, in case modems are used; but the direct connections delete those signals to minimize conflicts in their handling by individual communication ports on different vendors' equipment.) The connection must be supported in full-duplex mode, i.e. each station must be prepared to receive while transmitting.

The associated link layer protocol is AMRF-originated, providing frame definition and integrity checking only. It bears (intentionally) a weak resemblance to IEEE 802.2. It is defined in the following paragraphs.

The transmitting entity wraps each service data unit received from the local network layer in an envelope producing this frame structure:

SOH, UI, LN0, LN1, SDU, CK0, CK1

where:

SOH	is the single byte with hex value 81, designating start-of-frame;
UI	is the single byte with hex value CC, designating "Unacknowledged Information" as in IEEE 802.2;
LN0	is the low-order byte, and
LN1	is the high-order byte of the positive binary integer designating the length of the SDU portion of the frame;
SDU	is the service data unit received from/presented to the network layer;
CK0	is the 0-byte, and
CK1	is the 1-byte of the "Fletcher checksum" of all bytes of the frame, from SOH to the last byte of SDU, inclusive.

The transmitting entity then sends this frame on the link associated with the intended receiving station, at the first opportunity provided by this interface, and treats the transmission as complete.

The receiving entity identifies the beginning of a new frame by the occurrence of the SOH and UI bytes, and otherwise discards received bytes until this pair is identified. The receiver then assembles the SDU length from the LN0 and LN1 bytes and, if the result is negative or zero, discards the frame and searches for a new frame identification. Otherwise, the receiver reads the number of bytes indicated by the length of the SDU field and two more for CK0 and CK1. If there is a substantial delay between receipt of any of these bytes, the frame is discarded and the search for a new frame resumes. Otherwise, when all of the bytes have been received, the receiver executes the Fletcher checksum algorithm [22] on all bytes received, from the SOH thru CK1. If the result is zero, the SDU portion of the frame is presented to the network layer. If the result is nonzero, the frame is discarded and the search for a new frame resumes.

This technique provides for frame definition and integrity checking with the "best effort" philosophy: Erroneous frames and lost frames are discarded entirely by the link layer service, without retransmission mechanisms, so that only correct frames, but not necessarily all frames, reach the receiving network layer. This is analogous to the data link protocols employed by the IEEE 802 standards. It assumes the availability of transport layer protocols to recover and retransmit lost information units.

2.1.2. Final Alternative 1: Broadband Token Bus

The AMRF network did not initially envision direct connection of every controller to a token bus, since there were no standards at that time and no implementation could be expected to be available for more than a few computer or controller systems. The MAP effort, which created and adopted the IEEE 802.4 Broadband Token Bus [18] standard, has made this more likely although not currently possible since commercial products are not available for all AMRF computer systems.

In the interim, the AMRF acquired (in 1984 before the adoption of the IEEE standard) a non-standard commercial broadband token bus network. Because of delays in the network installation and conflicts with MAP (see below), no AMRF controllers use the MAP protocol at the moment. Meanwhile, this broadband token bus network is used to support the AMRF network architecture by providing transparent services between computer systems separated by a distance of 1 kilometer.

2.1.3. Final Alternative 2: Baseband CSMA/CD Bus

The physical and data link protocols are defined by IEEE 802.3 Local Area Networks: Carrier Sense Multiple Access with Collision Detect [6,9], specifically the 10 MHz baseband option. These protocols define a common bus on which any connected station may place a message for any other connected station at any time. The stipulation of CSMA/CD is that a station detects an existing message-in-progress (Carrier-Sense) and does not interrupt it. Since two stations waiting for the same ongoing message to finish may simultaneously initiate new messages, the possibility of accidental "collision" exists and must be accounted for (Collision Detect) and both stations must "back-off" and retry later.

The Xerox 10 MHz Ethernet (versions 1 and 2) is nearly identical to this protocol and may operate on the same physical bus, but the data link layer is just enough different in format to be incompatible. Fortunately most Ethernet stations can communicate with 802.3 stations after a small software revision, so an Ethernet interim interface can usually be converted to a "final" IEEE 802.3 interface in the field.

The AMRF envisioned this as the typical engineering or administration network and the only available standard for a local high-speed network for real-time control. The expectation was that there would be multiple such networks linked together by "gateways" (MAP calls them "routers") on the broadband token bus. At present many AMRF controllers are locally networked by CSMA/CD networks, and the integrating gateways (which are present) are unused.

2.1.4 Final Alternative 3: High-Speed Bus Link

The alternative approach to integrating systems into the global broadband network was to construct local high-speed point-to-point links to a network "front-end". The approach uses an RS449 synchronous serial interface at one of two speeds to link the computer/controller system to a "Network Interface Unit" which would itself be directly connected to the token bus.

The physical layer is defined by EIA RS449 [19] full-duplex synchronous point-to-point modemless connection at 56 Kbps (or 500 Kbps) using the following circuits: Send Data (SD), Receive Data (RD), Send Timing (ST), Receive Timing (RT), Terminal Ready (TR) and Data Mode (DM), which require Send Common (SC) and Receive Common (RC). The connection must be supported in full-duplex mode, i.e. each station must be prepared to receive while transmitting.

The data link layer is defined by ANSI X3.66-1978 High Level Data Link Control Protocol (HDLC) [4] for "asynchronous balanced pair", using subset of data unit types defined by CCITT X.25 LAP B. This is the common implementation of "HDLC" offered by many vendors to support connection to public networks.

In the AMRF original view, this "front-end" protocol was to be treated as a pure subnetwork protocol (even though each such subnetwork would have exactly two stations) mandating the use of the Network layer protocol to effect connection to the target host, in the same way as the IEEE 802.3 subnetworks above.

Fortuitously, MAP adopted this identical protocol for the "interim interface" protocol to be used with its NIUs. Unfortunately, the MAP token bus implementor layered a nonstandard asymmetric flow-control protocol on top of the ANSI/ISO HDLC protocol. Further, because MAP did not originally accept the subnetwork concept, the perception of the NIU as an internetwork link was not accepted either. In the MAP interim interface, the NIU "exposes" the IEEE 802.4 link layer, so that the host system constructs and receives IEEE 802.4 frames enveloped in the HDLC plus nonstandard flow-control protocol frame.

In order to use the available commercial products, the AMRF network is forced to convert to this approach. At this time, none of the AMRF stations is so connected.

2.2. Network Layer

The AMRF network specifies the ISO 8473 Connectionless Network Service Protocol to provide for host-to-host message routing services. Briefly, this is a "datagram" protocol, in which each data unit is labelled with the sending and receiving host and finds its way through the network from the sender to the receiver without regard to any previous or concurrent transmissions.

This permits the introduction of "internet gateways" (what MAP calls "routers") - stations on two or more networks which receive messages on one of the networks and retransmit them over another of the networks toward the destination indicated in the network layer envelope. Since the AMRF network is intrinsically a multi-network involving several separate physical protocols (with associated data link protocols), any of the hosts can be connected to more than one network. If such a host receives a data unit for which it is not the indicated destination, it can function as a gateway by retransmitting the data unit by whatever means it would have used to reach the indicated destination. In addition, "gateway stations" have been procured to link the major subnetworks to the token bus.

Because the destination is clearly specified in the data unit, a receiving station can determine whether the data unit is intended for that station or must be relayed to another; and because the source is clearly specified, the destination station can determine the true originating host regardless of the path over which the data unit arrived. To make this simple mechanism work globally, the AMRF network specifies the network layer protocol as mandatory, even when a point-to-point link or direct connection to a common bus is used. The network layer software is then required to support multiple underlying physical/link protocols (most stations have serial connections and bus connections) but not to understand any qualitative differences between them.

2.2.1. Interim Network Layer

Some AMRF systems (notably the SUN workstations) come equipped with Ethernet (or IEEE 802.3) network connections, but do not support the ISO protocols on that network, or at least do not support ISO protocols on the same network interface on which they depend for their "private" services. These stations characteristically use the MilSpec-1777 Internet Protocol [20] instead of ISO 8473.

MilSpec-1777 is functionally equivalent but totally incompatible in representation. The AMRF currently tolerates this protocol on certain subnetworks, but this requires a much higher level "gateway" service to provide communication between those subnetworks and the rest of the facility.

2.3. Transport Layer

In the area of transport protocols, the AMRF network is in a state of change. The nominal standard identified in 1984 - the ISO standard, which coincided with the MAP choice - had no commercial implementations available until 1986. This necessitated the implementation of an interim transport service protocol for the engineering phase of the AMRF, which was then used from 1983 through 1986. Moreover, the AMRF network architecture anticipated the use of a single global transport protocol, which made it difficult to accommodate adoption of the standard on some stations while it was still unavailable on others.

Recently, it has become necessary to accommodate yet another interim transport layer protocol, in this case the military standard adopted in 1983, now available on many commercial systems, often to the exclusion of other protocols. The current method of accommodation is to implement identical application layer services with two separate underlying network service sets (see Topology), which is clearly a short-term solution.

While the goal is still the global standardization of the ISO protocol, it is not clear whether that will occur soon enough to preclude revision of the AMRF network application and session layers to support multiple transport layer protocols.

All of these protocols essentially provide the same three services:

1) end-to-end integrity checking, message ordering and retransmission, guaranteeing that every message reaches its destination and messages arrive in the order they were sent;

2) end-to-end flow-control, allowing stations to control the rate at which data is transmitted to match the rate at which it can be processed;

3) segmentation and reconstruction of data units, allowing messages of arbitrarily large size to be transmitted, by breaking them into blocks convenient for the network medium.

The individual protocols are identified below.

2.3.1 Final Standard Transport Protocol

The nominal AMRF standard is ISO 8073 Transport Layer Service Protocol [21] Class 4. The class distinctions restrict transport services, and thereby complexity, according to the degree of simplicity and reliability provided by the lower layers. The AMRF multi-network environment mandates class 4 services - the highest class, which assumes little reliability in the lower layers - but, because it is a local area network, very few of the extended options. This is essentially identical to the MAP transport protocol selection.

While this protocol is implemented on several of the AMRF stations, it is not currently in use. This is because of the "multiple transport protocols" problem indicated above.

2.3.2 Interim Standard Transport Protocol

The interim standard is the Transmission Control Protocol for Defense Networks (TCP), MilSpec-1778 [7]. This protocol is not strictly a "transport" protocol in the OSI model sense, since, in addition to transport functions, it contains a limited session management protocol and a primitive application selection protocol (which used to be thought of as a session-layer function) as well.

This standard is used in the AMRF network only on those hosts which use this protocol as part of a large class of distributed services offered by the manufacturer and do not support any other protocol (simultaneously) over the principal network. Use of this protocol, and thus the principal network, affords efficient communication among these stations, while use of the alternative serial asynchronous links (where the other protocols may be supported or implementable) affords very poor communication services. At least one primary network station must implement this, as well as the AMRF standard protocols in order to enable interchange between these stations and all the stations implementing the standard protocols.

2.3.3 Interim Local Transport Protocol

The interim transport protocol currently in use on serial links and some Ethernets is a 1982 AMRF design providing the common transport services, originally intended as an interim until a standard should be developed. It has outlived all expectations, largely because of difficulties encountered in implementing and acquiring implementations of the ISO standard protocol. This protocol is defined in detail in Appendix G.

2.4. Session Layer

The AMRF network has a nominally "void" session layer. Unlike the few existing ISO application layer standards, the operating AMRF application layer service (common-memory mapping) is a station-to-station service which is itself a multiplexer. That is, the single mapping-service to mapping-service connection may (unwittingly) carry any number of logically separate communications. Thus the "session" layer is subsumed.

2.5 Presentation Layer

The AMRF network currently has a "void" presentation layer. All control interchanges are in some form agreed to by the parties involved. It is expected that the incorporation of the distributed data protocols in a future version of the network architecture will result in the standardization of some interchange form for all message units, which may obviate presentation layer protocols indefinitely.

2.6 Application Layer

The AMRF network currently provides only one "application service" at all stations: the "memory mapping service". The memory mapping service is the means by which the "common-memory" concept is extended to multiple computer systems.

The common memory architecture is described in Section III and the mapping protocol is defined in Appendix H of this document.

3. NETWORK INTERFACE PROCESS (NIP)

3.1. General Description

The Network Interface Process (NIP) is the software element of the AMRF network that logically interfaces the processes at one physical station to the AMRF network and any other process on it.

The "direct" interface between any two processes in the AMRF, regardless of residence, is referred to as a "mailbox": The writer inserts a message into the mailbox, and the reader retrieves the message from it. The mailboxes used by a process are always physically located somewhere on the station where the process itself resides. When a message must be transferred between processes residing on two different network stations, the NIPs at the two stations cooperate to copy the message, via the network, from the writer's mailbox to the reader's mailbox. The sole purpose of the NIP is to implement these transfers. All of the individual NIP functions are simply components of this task.

The general rule is that every mailbox has exactly one writer, although it may have several readers. The NIP itself is the writer for every local mailbox which is to be filled by a process elsewhere on the network, and the reader for every local mailbox which is to read by a process elsewhere on the network.

The NIP acquires its knowledge of which mailboxes to receive and which to transfer from the network manager and keeps this information in an internal data structure called a mail delivery table. It also reports, to the network manager, any problems encountered in maintaining these connections. (Section IV.4 describes the network manager in detail.)

The NIP uses local mailboxes according to direction from the network manager. It is not involved in the assignment of mailboxes; the assignment of mailboxes is a function of the network manager and the local Mailbox Manager.

The NIP is divided into two sections: the Networking section, which handles the networking device and the network protocols, and the Interfacing section, which handles the local mailboxes and the connection table.

The NIP is designed to effect the transfer of messages between mailboxes on the network in a fashion totally invisible to the other processes on the station. User processes must be able to communicate with each other without knowing whether they are on the same station or different stations. Therefore the contents of a mailbox must not be altered in any way by its transit through the network.

The NIP communicates directly only with the other elements of the communications system: the network manager and the local Mailbox Manager. User programs do not deal directly with the NIP; they communicate with the network manager and the network manager communicates with the NIP.

3.2 Network Device Driver

The NIP operates the network interface device on each station. It initializes and maintains the device, directs the device to receive all network packets intended for this station, and directs the device to transmit each packet outbound from this station.

3.3. Protocol Implementation

The NIP executes the link, network and transport protocols at this station, with the assistance of the device firmware. It implements an access control protocol, with the assistance of its firmware interface, in which it competes for authority to transmit. It constructs network packets for outbound messages, and extracts inbound messages from network packets. It acknowledges successfully received packets and ignores corrupted ones. It implements transport controls to handle receive buffer shortages at either end of a connection.

3.4. Session Control

The NIP maintains an internal connection table specifying:

(1) which local (common-memory) mailboxes are to be transmitted

(2) to which address on the network and under what conditions

(3) which local mailboxes the NIP will fill from messages on the network and from which address those messages will come.

This table is initialized to contain only NIP to network manager mailboxes. Additional entries in the connection table are made under the direction of the network manager.

The NIP obtains control of the station at station reset and initializes the network interfacing process and the local mailbox manager at that time.

When the NIP receives a message, it looks up the address of the sender in its connection table and places the message into the local mailbox specified in the connection table entry.

During its processing cycle, the NIP examines each of the local mailboxes for which it has a transmission entry in its connection table, determines whether the mailbox is eligible for transmission and if so, sends it out on the network to the destination identified in the connection-table entry.

3.5. Error Reporting

The NIP informs the network manager of any network performance anomaly it detects. There are three kinds of error reports:

(1) Device Error: reported when the network interface device indicates that a local error has occurred, or fails to respond to an operation;

(2) Link Error: reported when the remote station fails to acknowledge a message after the configured maximum number of retries, even though the interface device did not indicate a failure;

(3) Remote Error: reported when the NIP successfully receives a message for which it has no connection table entry, and therefore no mailbox to put it in.

3.6. Configuration Parameters

Each NIP must have the following information parameterized in its source code and set at compilation or binding time:

1. the Local Station Address. The NIP must know its own address so that it can initialize its network interface to match messages intended for it.

2. the Station Address and Subaddress of the network manager.

3. the Local Subaddress for Manager-to-NIP transmissions, always one (1). Effectively, this is the "command input" mailbox for the NIP.

4. the Local Subaddress for NIP-to-Manager transmissions, always zero (0). Effectively, this is the "status output" mailbox for the NIP.

5. the mapping algorithm for translating between network subaddresses for this station and local mailbox identifications.

The network manager interface parameters are required for proper initialization of the NIP connection table via directives from the network manager. The related "internal" mailboxes are not required to be implemented in the same fashion as the rest of the local mailbox architecture. These mailboxes are used for information transfer only between elements of the NIP itself and are local storage to the NIP. They are required to appear to be mailboxes only because they are visible subaddresses on the network and every subaddress corresponds to a "mailbox".

3.7. Connection Table Entries

Connection table entries have the form:

Local Mailbox,
Remote Address,
Mode,

where Local Mailbox is a number specifying which local interchange mailbox or common-memory buffer is to be used; Remote Address is the network station address and subaddress to be attached to that mailbox; Mode specifies whether the local mailbox is to be written from messages received with the given Remote Address or transmitted to the given Remote Address whenever it changes.

Connection table entries are constructed by directives from the network manager.

3.8. Directives

NIP directives consist of instructions for connection table maintenance: Add-Entry, Delete-Entry. These directives come from the NIP's initialization procedure and from the network manager.

3.9. Mailboxes To Be Transmitted

Mailboxes to be transmitted must have the local standard form of interprocess communication mailboxes (common-memory buffers, etc.). In each case, the contents of the mailbox must in a standard way indicate whether the mailbox contents has changed since the last time it was read by the NIP. A standard location in every mailbox text must provide a sequence number which is updated each time the mailbox has a "new" content.

3.10. Network Packets

Incoming data are in the form of packets on the network which comprise a mailbox message, packaging and integrity check envelopes, and a routing envelope. The routing envelope contains the destination address which decodes into a station and subaddress. The subaddress, in association with a corresponding connection table entry, identifies the local mailbox.

The NIP will extract the message from incoming network packets and write it into the corresponding local mailbox via the appropriate local mailbox protocol. The text delivered into the mailbox will be the image of the source mailbox contents; there are no additions or substitutions made by the NIP.

Network Packets will be constructed from outbound local mailbox messages, by addition of a routing envelope per HDLC (ANSI X3.70-1978), giving station and subaddress from the connection table entry for the local mailbox, and a link envelope per SDLC.

3.11. Initially-Given Mailbox Connections

Every process is started with at least two given mailboxes, which are the command and status (response) mailboxes for communication with the network manager. In an implicit transfer system, these are associated with fixed user buffers; in an explicit system these are associated with fixed user mailbox identifier variables. For this document, they will be designated NIP_CMD and NIP_STS respectively.

Additionally, a mail delivery table entry is made to connect the NIP_CMD mailbox to the network manager system. This mailbox serves a "bootstrap" function for the NIP, since all further CONNECT commands arrive in it, including the command to connect the NIP_STS mailbox for output to the network manager system.

4. NETWORK MANAGER (NETCMD)

4.1. Description

NETCMD is not a true network manager. Instead, it is an operator interface designed to allow the human network manager to examine and modify the configuration and status of the network. The operator enters commands in a legible syntax. NETCMD converts these into the required data structures and enters the resulting NIP commands into the correct mailboxes for the Network Interface Processes.

4.2. The Network Manager Display

This display shows the current status of the network, giving a formatted display of the NIP status for each NIP, as last reported by that NIP (Figure IV-1).

4.3. Network Manager Commands

4.3.1. CONNECT And DISCONNECT Mailboxes

The format for the CONNECT and DISCONNECT commands is shown in Figure IV-2. The command encoding scheme is described below.

```
                         VAX                HWS  HMC  HMB  HRC  HGP
NIP status                UP                 UP   UP   UP   UP   UP
MDT entries               44                 8    4    4    8    6
cmd #/status            45/1=OK             8/1  4/1  4/1  8/1  6/1
time since              00m:00s             0000 0000 0000 0000 0000
               ------------------------ ---- ---- ---- ---- ----
to station     HWS  HMC  HMB  HRC  HGP  VAX  VAX  VAX  VAX  VAX
status         NORM NORM NORM NORM NORM NORM NORM NORM NORM NORM
error code        0    0    0    0    0    0    0    0    0    0
mgrams in       156  135 1419  164  110  936   14   17   29    7
mgrams out      936   14   17   29    7  155  134 1417  162  109
retransmit        0    0    0    1    0    0    1   14    3    4
poll count        0    0    0    0    0    0    0    0    0    0
IOA             400  766  311  455  677  033  666  122  533  755
xport state                        XMIT



Command:
```

Figure IV-1. Network Status Display

Where the command fields in Figure IV-2 are separated by one or more blank characters, some sort of delimiter must be used. This can be white space, a tab, or a comma. The significance of each field of the command is described below, identified by its "field number".

1. The first field indicates the station to which the command is to be send. If the command is CONNECT, an entry will be made in that stations mail delivery table (assuming the remaining fields are valid). Likewise, if the command is DISCONNECT, an existing entry will be flagged to indicate that the connection no longer exists. The station name corresponds to the site identifier found in source code listings netgen.c68, netcmd.c and *def.a68.

2. This is the first part of the action field. It specifies whether to 'c'onnect or 'd'isconnect the specified mailbox.

```
 1-- name of station to which this cmd is directed
 ¦
 ¦    2-- Connect or Disconnect
 ¦    ¦
 ¦    ¦    3-- Input, Output or Duplex (both)
 ¦    ¦    ¦
 ¦    ¦    ¦      4--mailbox name
 ¦    ¦    ¦      ¦
 ¦    ¦    ¦      ¦    5-- swap flag (if the bytes are
 ¦    ¦    ¦      ¦    ¦   in Intel order put an * here)
\/   \/   \/     \/   \/
dst {C¦D}{I¦O¦D} mbx [swap]length [type] station sockid [addr]
                            /\      /\     /\     /\     /\
                             ¦       ¦      ¦      ¦      ¦
     length of the mailbox --6       ¦      ¦      ¦      ¦
                                     ¦      ¦      ¦      ¦
 mailbox type (if NIP CMD/STS mbx) --7      ¦      ¦      ¦
                                            ¦      ¦      ¦
                   name of remote station --8      ¦      ¦
                                                   ¦      ¦
                socket ID (identifies connection) --9     ¦
                                                          ¦
                     address of mailbox (if applicable) --10
```

Figure IV-2. NETCMD Command Structure for Connecting and Disconnecting Mailboxes.

3. This tells the NIP whether the connection is 'i'nbound, 'o'utbound or both ('d'uplex, bidirectional).

4. The mailbox name must be placed in this field. This is expected to be the same on both sides of a connection.

5. The swap flag, an asterisk, is placed at the beginning of the length field if the bytes on the station where the mail delivery table entry is being made stores its integers with the bytes in Intel order. Note: the VAX uses Intel order.

6. Length of the mailbox.

7. This is an optional field which indicates the mailbox type or structure if the mailbox functions for NIP command and status. The network software only understands two types of NIP mailboxes: nip_cmd (1) and nip_sts (2).

8. The name of the station on the other end of the mailbox connection.

9. This field contains the label (socket or session identification label) that uniquely identifies the connection between the two stations and is associated with the specified mailbox name. See Appendix C for instructions on creating these labels.

10. The address field is used when entering an entry into any site's table except the site which has the network manager (NETCMD) on it. It has meaning only on systems which have direct memory mapped mailboxes. On those systems, it is used as the starting address of the mailbox described in this entry.

4.3.2 Other commands:

Table IV-1 lists the remaining NETCMD operator commands. Those identified as "unsupported" are currently not implemented. Detailed description of commands are given below.

Comments are for readability in command files. When "!" is used the comment will be displayed by netcmd while the commands are being executed. Comments with periods are not displayed.

A "?" or "H" will cause all the commands to be displayed, even the unsupported ones.

A "@" will cause what immediately follows to be interpreted as a file containing netcmd commands to be executed. VMS paths will be interpreted correctly when included in the file name.

"K" <station> will cause the VAX NIP to disconnect all its mailboxes to the specified station.

Use "Q" to quit netcmd.

The "Z" command will zero out the command mailbox for that station. It sets all the fields except the sequence number to zero. The syntax for the command is <station> Z.

A "(ctrl) L" will rewrite the screen, this is an important feature when running netcmd on a non-VT100 screen.

"(ctrl) M" or <CR> causes netcmd to update the data on the display. This is useful when watching the "time since" field for a station.

!	comment (displayed)	supported
.	comment (not displayed)	supported
?	help listing	supported
@<filename>	read cmds from filename	supported
H	help listing	supported
{K ¦ L} <station>	disconnect or reconnect link to station	semi-supported
P	send no-op command to nip	unsupported
Q	quit netcmd	supported
R	direct nip to resume bus operation	unsupported
S	direct nip to suspend bus operation	unsupported
X	direct nip to halt/exit	unsupported
Z	clear nip command mailbox (except sequence number)	supported
^L	refresh screen	supported
^M	update display	supported

Table IV-1. NETCMD Operator Commands. Single-character commands can occur in either upper or lowercase.

4.4. Communications To The NIP

The network manager communicates with the NIPs using the standard mailbox interface. NIP commands are mailgrams inserted by the network manager into its local common-memory and delivered by the local NIP to the intended recipient NIP. Each NIP reports its status into a local mailbox. That mailbox is delivered to the network manager by the NIP according to a local mail delivery table entry.

4.4.1. Command Structure

The command structure is as follows:

```
Length:  2-byte integer - length of this mailgram in bytes;
Seqno:   2-byte integer - command (mailgram) sequence number;
Time:    4-byte integer - mailgram time stamp;
Command: 1-byte character - nature of command, values:
                "C" = Connect
                "D" = Disconnect
                "N" = No Operation
Filler:  3-byte character - ignored, used to align fields
MDTent: 48-byte structure - skeleton mail delivery table entry;
Node:   32-byte character - name of network node in ASCII;
```

Except for No Operation, the NIP looks up the "node" name in its local port identification table and substitutes the corresponding internal port identification into the "network address" field of the "MDTent" structure and clears the dynamic fields of the MDTent structure to logically complete the structure.

4.4.2. Status Structure

The status structure is as follows:

```
Length:  2-byte integer - length of this mailgram;
Seqno:   2-byte integer - mailgram sequence number;
Time:    4-byte integer - mailgram time stamp;
CmdSeq:  2-byte integer - sequence nr of last command received;
Status:  2-byte character - completion status of last command:
           "OK" = command completed successfully;
           "NG" = command in error - not completed;
Ecode:   4-byte integer - code for type of error in last command;
MDTect:  2-byte integer - number of entries in the mail delivery
           table;
Nports:  2-byte integer - nr of ports for which status reported;
Pstat:  16-byte structure for each port, comprising:

        Portid:  2-byte character - port/line identification;
        Channel: 2-byte integer - local system identification
                 for the line;
        InStat:  2-byte integer - status code for last input
                 operation;
        OutStat: 2-byte integer - status code for last output
                 operation;
        Mode:    1-byte integer - operational mode of the link,
                 values:
                       0 = DISC - disconnected
                       1 = INIT - initializing
                       2 = NORM - normal
                       3 = SYNC - waiting for output completion
                       4 = SHUT - shutting down
```

	5 = ERR - error on link
	6 = NRSP - no response, too many polls
	7 = CREJ - remote refused connection
	8 = NOSY - noisy line, too many packet errors
Iseq:	1-byte integer - next input sequence number expected on link;
Aseq:	1-byte integer - last acknowledgement sequence number received;
Oseq:	1-byte integer - next output sequence number to be used;
PollCt:	1-byte integer - count of consecutive unanswered polls sent;
Noise:	1-byte integer - count of consecutive bad packets received;
RcvSts:	1-bit logical - receiver status: 0 = local receiver ready; 1 = local receiver not ready
XmtSts:	1-bit logical - transmitter status: 0 = remote receiver ready; 1 = remote receiver not ready
AckReq:	1-bit logical - 1 if acknowledge must be sent
PollReq:	1-bit logical - 1 if poll must be sent
DataReq:	1-bit logical - 1 if data waiting to be sent
XmtCmd:	1-bit logical - 1 if transmitting a command
PollWt:	1-bit logical - 1 if waiting for poll response
XmtAct:	1-bit logical - 1 if transmitter active

4.5. Interface To VAX Common Memory (MBHAND)

4.5.1. Description

As currently written, the NETCMD program on the DEC VAX 11/785 utilizes a mailbox handling interface utility, called MBHAND, to exchange mailgrams with common memory. The purpose of the interface is to allow processes which have to wait on external events, e.g. terminal activity, to access common-memory mailboxes without stalling other, higher-speed processes. Only NETCMD uses MBHAND.

The mechanism of communication between NETCMD and MBHAND is the VAX/VMS interprocess mailbox MBX_HANDLER. NETCMD inserts commands into MBX_HANDLER. In general, there is no status feedback. In the case of the Mailbox Read command, one of the command fields associates another VMS mailbox to receive the current mailgram in the designated AMRF mailbox. NETCMD determines which VMS mailbox is to be used.

4.5.2. MBHAND Commands

The following subsections describe the MBHAND commands.

4.5.2.1. Mailbox Connect

```
Command structure:
   type:  1-byte character, value "C" - indicates a Mailbox
                 Connect command;
   dir:   1-byte character, values:
                 "I" = input mailbox
                 "O" = output mailbox
                 "S" = input/output mailbox
   size:  2-byte integer  - size in bytes of the mailbox to be
                 connected;
   name: 32-byte character - name of the mailbox to be connected;
   vmbx: 32-byte character - name of the VMS mailbox to receive
                 the contents of the designated common memory
                 mailbox.
```

The common memory mailbox identified by "name" is opened for input or output per the "dir" field with the specified "size". The common-memory variable need not be a mailbox. If "dir" is "I" and "vmbx" is non-null, MBHAND attaches the VMS mailbox designated by "vmbx" and associates it to the common memory mailbox designated by "name". A common memory mailbox should be connected before it is referenced in a Mailbox Read or Mailbox Write command, in order to correctly set the size of the mailbox and direction of reference. If a common memory mailbox has not been referenced by a Connect command, MBHAND will open it on the first reference, using the size and direction specified in the first reference.

4.5.2.2. Mailbox Write

```
Command structure:
   type:  1-byte character, value "W" - indicates a Mailbox
                 Write command;
   dir:   1-byte character, value <NULL> - not used;
   size:  2-byte integer - size in bytes of the mailgram
                 to be written;
   name: 32-byte character  -  name of the common-memory
                 mailbox to be written;
   text: up  to  520-byte data  structure  -  mailgram
                 to be written.
```

"Size" bytes of the specified "text" are written to the common memory mailbox identified by "name". If the mailbox was not previously connected, it is opened for output by MBHAND.

4.5.2.3. Mailbox Read

Command structure:

type:	1-byte	character, value "R" - indicates a Mailbox Read command;
dir:	1-byte	character, value <NULL> - not used;
size:	2-byte	integer - size in bytes of the mailgram to be read;
name:	32-byte	character - name of the common memory mailbox to be read;
vmbx:	32-byte	character - (optional) name of the VMS mailbox to receive the contents of the designated common memory mailbox.

"Size" bytes of the specified common memory mailbox identified by "name" are written to VMS mailbox designated by "vmbx". If the mailbox was previously connected with "vmbx" specified, "vmbx" may be null, and the VMS mailbox associated by the Connect command will be used. Otherwise "vmbx" is required. If the variable was not previously connected, it is opened for input by MBHAND.

5. SUBNETWORKS

Subnetworks are used within the AMRF for two reasons:

(1) To maximize response time (the only traffic on the subnetwork is the traffic common to those stations).

(2) To minimize network loading (no unnecessary traffic echoed throughout the cable plant and generating unnecessary loading.)

Two subnetworks are identified in Figure IV-3:

(1) at the Inspection Workstation (serial RS232), and
(2) at the Horizontal Workstation (Ethernet).

Real-time control dialogue can overload the plant network. The purpose of the subnetworks is to isolate large volumes of local traffic from the primary network pathways. This has the advantage of enhancing overall communications performance.

The subnetworks are connected into the main plant network through "gateways."

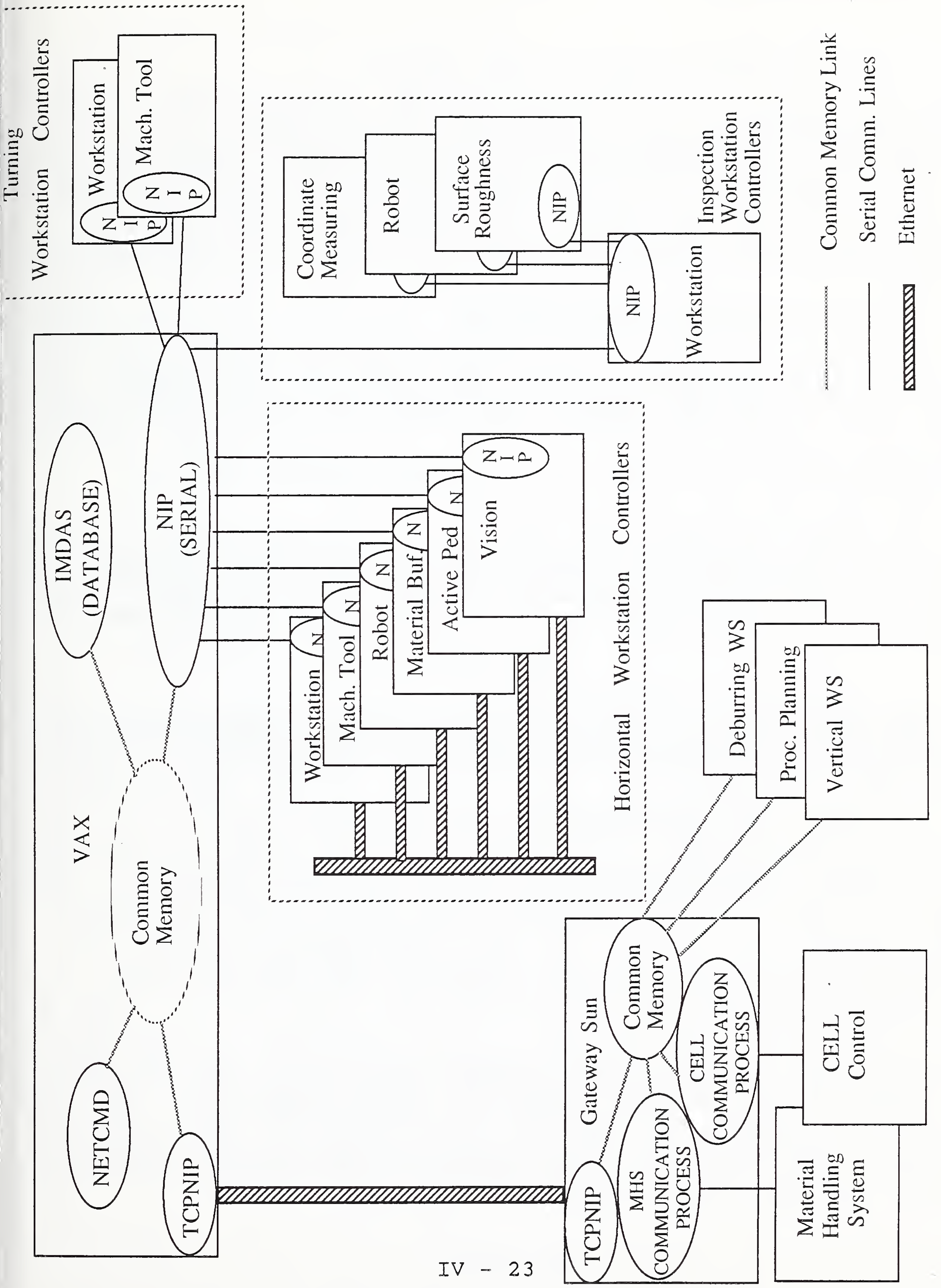

Figure IV-3. The Topology of the 1986 AMRF Network

6. SECONDARY COMMUNICATIONS SYSTEMS

Some point-to-point connections that do not implement AMRF network protocols are used to provide interim communication links between the local common memory of controllers that are not on the AMRF network and the global AMRF common memory. These temporary communication services will soon be replaced with a MAP broadband interface and direct links to the AMRF global common memory and Integrated Manufacturing Data Administration System (IMDAS).

6.1. <u>PC-to-Sun (CELL and MHS)</u>

Two temporary interfaces are currently implemented to link the CELL and material handling system (MHS) PCs to the global AMRF common memory through a local common memory interface resident on a Sun microcomputer workstation. This temporary interface consists of a serial asynchronous module that communicates with the respective PC and a TCP module that uses sockets to communicate with the local common memory server.

In the future, as appropriate software and hardware become available, this temporary interface will be replaced. A common memory system and network interface process (NIP) providing direct access to the AMRF MAP network and the AMRF global common memory will be installed on each of the PC-based controllers.

6.1.1. TCP Communications

The TCP communications architecture has already been described in Section III.2.2.1. Programmers reference information is available in Section V.1.2.

6.1.2. The Serial Communications Link

The transfer of messages across the serial link is coordinated using a master-slave relationship between the communications module on the Sun and the communications module on the respective (CELL or MHS) PC. The PC is designated as the master and the Sun is designated as the slave. The master has the time critical process running on it, and thus determines when message traffic can be sent over the link.

The master and slave serial port servers coordinate the transfer of messages by a protocol which depends on sending. The protocol is composed of byte counts, communications control blocks, and data message blocks. The byte count is used to tell the system on the other side of the link the number of bytes to read from the port. The receiving system echoes the byte count to indicate that it is ready to receive the next message.

6.1.2.1. Message Structure

A communications control message is typically five bytes in length. It is used to set system mode to send or receive all data message blocks, indicate end of mode (i.e., all incoming or outgoing message blocks have been sent), acknowledge or negative acknowledge a valid message was received (i.e., initiate retransmission of bad blocks), etc.

The communications control blocks and data message blocks have the same four byte header consisting of a checksum, a message length, a mailbox number, and a message block number. Communications control blocks have a one byte text field that indicates a change in communications mode, or an acknowledge or negative acknowledge for the last block. Data message blocks have a one to 240 byte text field that may contain a whole message or one block in a chain that together comprise a whole message.

6.1.2.2. Module States

The different stages in the communications cycle are implemented as finite state machines. When the communications module is activated by the system supervisor module, it cycles through the state machine performing communications functions until an EXIT state is reached.

6.1.2.2.1. Check Status State

When the communications module is activated, it first determines whether or not it is time to send or receive messages. Message transmission is currently set to occur on 7.5 second intervals for performance reasons. Because of the 9600 baud serial link to the Sun computer and the relative infrequency of traffic at the cell level in the AMRF hierarchy, this interval seems to be satisfactory. When the direct network link is established on the CELL controller, traffic will probably be processed on a much more rapid control cycle basis.

6.1.2.2.2. SEND State

If it is time to transmit messages, one of the following actions is taken. If the communications management module is activated in SEND mode, it checks a counter on each mailbox to determine whether or not it contains new outgoing mail. Each mailbox has a pointer to the chain of ready message blocks that make up the mailgram. Each message block has a four byte binary header and between one and 240 bytes of text. The header includes a checksum, a block length, a mailbox number, and the number of the particular block within the mailgram.

Pointers to all READY message blocks are entered into a READY blocks table. The communications manager places the communications server on the Sun microcomputer in RECEIVE mode. Message blocks are subsequently transmitted and acknowledged. The servers at both ends of the serial link automatically handle some error recovery and retransmission of garbled messages. The messages are transferred from the mailboxes internal to the server on the Sun computer to the Sun common memory areas by TCP subroutine calls. Once mailgrams reach this memory area they are accessible to the AMRF network and all other systems within the AMRF that are connected to the network.

6.1.2.2.3. RECEIVE State

If the communications manager is in RECEIVE mode, it uses the local protocol to place the server running on the Sun into SEND mode. The Sun communications server follows a procedure similar to that outlined above (Section 6.1.2.2.2) to transfer new mailgrams that it has obtained from common memory on the Sun. As each message is received on the PC, the communications module obtains a data block from a free list, copies the incoming bytes into the block, reads the header, performs checksum calculations, chains error-free blocks into the specified mailbox, and updates the appropriate data sequence numbers. If necessary, it will request retransmission of garbled blocks.

6.1.2.2.4. EXIT State

Once all messages from the Sun have been received, the communications module enters the EXIT state and control is returned to the system supervisor so that other CELL functions may be performed.

V. PROGRAMMER REFERENCE SECTION

In many cases, the common memory interface is tightly coupled with the NIP interface, and it is difficult to discuss one without concurrently discussing the other. The following sections describe the interfaces to common memory and the NIP. The description for the various implementations is presented in the respective separate section ONLY if there is a logical separation in the services provided. Where no logical separation exists, the descriptions for both are located in the NIP section.

It is assumed that the individual reading through this material is familiar with structured programming languages such as "C" and Pascal. The NIPs are primarily written in "C" (for all systems except the IWS) and Pascal (only the IWS). However, due to system language or service limitations, some code on virtually all systems had to be written in assembly language.

This material is to be used for reference information, and assumes that the reader is referring to a NIP source listing. The Multibus version of the network interface program is the most representative of all NIPs, since most other NIPs are derived from it. Likewise, the VAX common memory [14, 15, 16] is the most representative common memory implementation.

Since most NIPs are identical, no attempt has been made to provide a complete list of function calls, arguments, and returned values for the individual NIPs. Again, the reader is encouraged to examine the Multibus section for representative NIP information (the VAX TCP NIP shows representative interfaces for this other version of the NIP), and reference [15] for common memory interfaces. The remaining implementation descriptions identify program coding or structural differences from these original implementations.

1. COMMON MEMORY

1.1. DEC VAX 785 Implementation

The VAX common memory implementation is based entirely upon the work performed on the Hierarchical Control System Emulator (HCSE). All interface specifications are documented in the respective references [14, 15, 16].

1.2. Sun Microsystems Implementation

1.2.1. Introduction

This system emulates a shared memory system and is loosely based on the common memory implemented for the VAX through the Hierarchical Control System Emulator (HCSE). User processes can reside on the same computer system as their local common memory, or they can be distributed throughout a number of other remote computer systems accessible thru the TCP/IP-based local area network. Likewise, the shared memory system can reside entirely within a single computer system or can be distributed across several computers linked in the same way. A server, the common memory manager, handles requests to manipulate shared variables.

The Sun common memory emulation was originally designed for the purposes of providing communication between hierarchically controlled processes. There are a small number of functions specifically for the purpose of making communication of such style easier, but the emulator is certainly not restricted by this and, thus, it also provides communications between processes with arbitrary relationships.

The Sun common memory is implemented as memory private to the common memory manager, which it reads and writes in response to requests by clients (i.e., a centralized access control). The HCSE uses a distributed form of control, depending on each process to pass access control to the next. A defect of distributed control is that the unexpected death of a process halts the emulation when access control is passed to the dead process during the next cycle. By centralizing access control, the unexpected death of any user process will not halt the emulation. The common memory manager can be monitored and any another process can take over the responsibilities of a deceased process.

1.2.2. Required Processes

There is only one special process that must always be running before a user application attempts to attach to common memory: the common memory manager. The existence and/or activation of auxiliary processes such as front-ends, debuggers, and editors is not necessary for operation of the common memory and does not affect a user application attaching to common memory.

The common memory manager handles requests from processes to read/write common memory variables. Various other requests are possible such as dynamically changing the size, type or writer of a variable. This last function is very useful when a process agrees to directly take over a resource that can be passed around between processes.

1.2.3. How To Use The Common Memory System

The common memory manager emulates a shared common memory with specific features for supporting communication between hierarchically controlled prccesses. However, communicating processes need not be hierarchically controlled.

This document describes how to use the current implementation on the Sun Workstation (running Sun UNIX 1.x, 2.x and 3.x). It has also been ported to the Silicon Graphics Iris. Sample calls are shown for both "C" and Lisp.

1.2.3.1. Process Identification

Before any other calls to the common memory system, the process should identify itself to the system:

```
        int rc = cm_process_name("HWS","cmm_host",0);    /* C */

        (cm_process_name "HWS" "cmm_host" 0)             ; Lisp
```

Here, we have declared ourselves as "HWS". This name need not be unique, however when debugging, you will find it helpful if you have chosen different names for your cm users.

The second parameter to cm_process_name() is the name of the host on which the common memory manager is running. Note that while there may be a common memory manager running on the local machine, this function call permits you to select a specific local or remote common memory for attachment. The local machine may be designated either by its name or by a null string.

The third parameter to cm_process_name() is an integer which indicates the debugging level. 0 indicates no debugging. Larger values request more debugging information. For example, 2 will give you information about messages sent and received. 5 will generate information about individual common memory values being manipulated. With 10, you will get a veritable flood of information (that you almost certainly don't want) including things like memory allocation, variable copying, etc.

cm_process_name() also performs some necessary initialization of CM client data structures. cm_process_name() returns 0 if successful. Anything else is an error. A common error is that the common memory manager is not running.

Before a second (or any further calls to) cm_process_name() call, cm_exit() should be called. cm_exit() tells the common memory manager that you have exited the common memory environment. It also cleans up various data structures internal to the cm system.

```
        cm_exit();

        (cm_exit)
```

1.2.3.2. Using Common Memory Variables

The following subsections describe the interfaces to common memory as well as the sequence for accessing a common memory variable.

1.2.3.2.1. Declaring Variables

All variables must be declared before use. cm_declare() is used to declare common memory variables.

```
        date = cm_declare("date",CM_TYPE_STRING,CM_ROLE_XWRITER,
                          CM_PERIOD_FOREVER);

        (setq date  (cm_declare
                     "date"
                     CM_TYPE_STRING
                     CM_ROLE_WRITER
                     CM_PERIOD_FOREVER))
```

cm_declare() returns an object that can be used when referring to this variable in the future. This object can be stored into a variable declared as type cm_variable. If cm_declare() returns CM_BAD_OBJECT, the declaration has failed (an error message will be printed out explaining why). Declarations can fail for a variety of reasons (bad or conflicting arguments, no space left to store values, etc.).

Once cm_declare() has returned an object, this object should be used whenever referring to the variable. In the case of cm_declare, the first argument is almost always a string, while in all other functions, the variable identifier is almost always an object. cm_declare() also allows an object as its first argument. This is for the purpose of redeclaring (for example, the type of) a variable during runtime.

In the example above, "date" is declared to be a string. From C, all the built in types are available. Only one C structure is usable, to support arbitrary sized pieces of memory. (More on this below.) From Lisp, only strings and vectors can be passed. This will be expanded in the future. For more information, see Section V.1.2.3.2.2. below.

The third argument of cm_declare specifies access rights. The available access rights are:

```
        CM_ROLE_NONXWRITER       or CM_ROLE_NONEXCLUSIVE_WRITER
        CM_ROLE_XWRITER          or CM_ROLE_EXCLUSIVE_WRITER
        CM_ROLE_READER
        CM_ROLE_WAKEUP
```

These access rights can be combined by ORing. For example, the wakeup right is always combined with at least one of the others. "wakeup" causes the common memory manager to wake the process up whenever the variable is written by someone else.

Conflicting combinations should be avoided. If one process has declared a variable CM_ROLE_XWRITER, other processes are prohibited from any kind of write access to that variable. These are the only restrictions on variable access.

The fourth variable in cm_declare() provides for a timeout on each variable. This is only meaningful when a variable is being written. A variable times out if it has not been written in the last p time units, where p is the period. Once a variable has timed out, it can be redeclared by any other process (typically for changing the access to writable). Timeouts are meaningless for non-exclusively-written variables.

Periods are defined easily by the function, set_period().

```
        #include <sys/time.h>

        struct timeval period;

        set_period(&period,seconds,milliseconds);
```

Two predefined periods exist for convenience. period_infinite is an infinite period of time. period_zero is no time at all.

1.2.3.2.2. Variable Types

1.2.3.2.2.1. Predefined Types

The following types can be used:

```
        CM_TYPE_INT
        CM_TYPE_FLOAT
        CM_TYPE_DOUBLE (or TYPE_REAL)
        CM_TYPE_STRING
        CM_TYPE_LIST
        CM_TYPE_CHAR
        CM_TYPE_BIT (or TYPE_BOOLEAN)
        CM_TYPE_SIZED
```

Even though some of these types are stored identically internally (e.g. CM_TYPE_INT and CM_TYPE_FLOAT), they are used differently when machine-to-machine conversion occurs, therefore, it is important that they be used precisely.

CM_TYPE_LIST is for communicating with Lisp programs. Lists are stored like strings internally, however they have formatting rules that must be followed in order for the data to be meaningful in the Lisp environment. Data generated by Lisp will appear in dotted pair notation. For example, the list (a (b c)) in dotted pair notation is (a . ((b . (c . nil)) . nil). Similarly, you must specify expressions in this dotted pair notation.

CM_TYPE_SIZED provides for raw bytes that are to be transmitted without interpretation. In this case, the data must include a size, so that the cm system knows how big the object is. Structure psiz_data is used for this.

```
        struct psiz_data {    /* pointer-to-sized data */
               char *data;
               unsigned short msize; /* size of malloc'd space */
               unsigned short size;  /* size of used space */
        }
```

If msize is 0, the common memory system will allocate space using malloc whenever the common memory system passes a value to the user. Further, if msize is ever smaller than the incoming value, the pointer will be realloc'd and msize increased appropriately.

Note, however, that languages such as Lisp which do not believe in malloc, will not be able to use the autoexpansion feature, since the CMS may attempt to free a Lisp object, which would be a serious mistake. To thwart such attempts, psiz_data should be initialized to point at a data block that is as large as will ever be necessary. msize should be the size of this space.

The address of the psiz_data object is passed as a cm_value and may be used as an argument to cm_set_value and its relatives.

The structure psiz_data is predeclared in Lisp (via c-declare) along with corresponding access functions. For example, to declare and set the size element of the psiz_data structure called "foo":

```
        (setq foo (make-psiz_data))
        (setf (psiz_data->size foo) 17)
```

1.2.3.2.2.2. User-Defined Types

The original common memory design was to support user-defined types, but experience with other common memory systems have taught us that this is "a bad thing". There is no reason why the common memory should know the type of the data that it is storing.

In practice, types other than CM_TYPE_SIZED are rarely, if at all, used. Variables are then encoded and decoded in accordance with ASN.1 (X.409). This provides for structured types which are machine independent.

1.2.3.3. Reading And Writing Variables

Variables may be read and written with the following calls:

```
        cm_get_value(variable,value);
        cm_set_value(variable,value);

        (setq value (get_value variable))
        (cm_set_value variable value)
```

Variable must be an object that has been returned by declare(). Value is the address of a value of the appropriate type for the variable. For example, to store a date in the date variable declared above, we would say:

```
        cm_set_value(date,"Wed Dec  5 13:45:55 EST 1984");

        (cm_set_value data "Wed Dec  5 13:45:55 EST 1984");
```

Several specific functions exist for handling handshaking between superior and subordinate processes in a control hierarchy. Specifically, variables can be used for command or status. Status variables are identified by the system with command value they are associated with.

Variables which are command variables should be read and written with the following routines:

```
        cm_set_new_command_value(variable,value);
        cm_get_new_command_value(variable,value);

        (cm_set_new_command_value variable value)
        (cm_get_new_command_value variable value)
```

One utility routine is available for determining whether a new command has been received. cm_new_command_pending() returns TRUE or FALSE depending on whether a new command has been received.

```
        maybe = cm_new_command_pending(command_variable);

        (setq maybe (cm_new_command_pending command_variable))
```

When a new command has been received, cm_new_command_pending will return TRUE until cm_get_new_command_value() has been called, after which it will return FALSE. cm_get_new_command_value()

also returns TRUE or FALSE, depending upon whether it has detected a new command.

Status variables must be written with the routine, cm_set_status_value().

```
        cm_set_status_value(command,variable,value);

        (cm_set_status_value command variable value)
```

Status (and any other) variables may be read with the routine cm_get_value().

Two predicates are available that are of use to the superior process in determining which command a subordinate process' status is in response to.

```
        maybe = cm_status_equal(cmd_var,stat_var,s_value);
        maybe = cm_status_synchronized(cmd_var,stat_var);

        (setq maybe (cm_status_equal cmd_var stat_var s_value))
        (setq maybe (cm_status_synchronized cmd_var stat_var))
```

cm_status_equal() returns TRUE or FALSE, depending on whether or not the status variable, stat_var, has the value, s_value, and is in response to the command specified by cmd_var.

cm_status_synchronized() returns TRUE or FALSE, depending on whether or not the status variable, stat_var, is in response to the command specified by cmd_var. This is very helpful to the superior process in finding out whether the subordinate process is responding to the command.

1.2.3.4. Synchronization

Since the common memory and user processes are actually unsynchronized processes, it is necessary to synchronize them occasionally. The idea of calling cm_sync(), is to force all variables common to the user and common memory manager processes to have the same value.

```
        cm_sync(behavior);

        (cm_sync behavior)
```

cm_sync() takes one argument that allows several behaviors by the common memory manager. There are two sets of options.

The first specifies whether cm_sync() should wait for at least one set of variable updates (or any response from the common memory manager). The default is CM_WAIT. To poll and return immediately, use CM_NO_WAIT.

The second option allows one the ability to examine the one set of variable updates before it has (possibly) been overwritten by an immediately following set of updates. This is very useful if you have a variable which you expect to be written by multiple processes.

Selecting CM_WAIT_AT_MOST_ONCE allows you to read a variable's value before it is overwritten by yet another value from the common memory manager. This is useful, for server processes, which take requests off a queue. The default is CM_WAIT_FOR_ALL which simply returns the most recently written value.

These options should be combined with a bitwise OR operation. For example, to poll for at most one new set of variable values:

```
        cm_sync(CM_NO_WAIT|CM_WAIT_AT_MOST_ONCE);
```

However, it is expected that most clients will simply want to use:

```
        cm_sync(CM_WAIT);
```

cm_sync returns either 0 (normal completion) or negative numbers denoting an error (see the cm.h source file).

1.2.3.5. More About Variables

Variables are more structured than in the VAX HCSE. This allows easier control of variables. For example, handshaking between two levels of the hierarchy is automatically handled by command associations embedded in the variable structures. Variables are also tagged with their type, length, etc...

Common memory variables have the following attributes:

```
    /* handshake.h */
    char name[MAXVARIABLENAMELENGTH];
    unsigned int type;
    unsigned long count;     /* nth definition of this value */
    unsigned long old_count; /*    -- ditto --  when last read */
    unsigned short size; /* size (bytes) of uninterpreted data */
    struct timeval *period;  /* how long data is good for */
    struct timeval *time_stamp;   /* when last written */
    command_association command_association;/* command with which
                                 this variable is associated */
    command_association old_command_association; /* when last
                                                     read */
    char *data;              /* uninterpreted data */
```

Variables also have the following attributes (which are strictly internal to the common memory manager).

```
    char xwriter[PROCESSNAMELENGTH];  /* exclusive writer */
        struct {
                unsigned reader : 1;
                unsigned writer : 1;
                unsigned wakeup : 1;
                unsigned new : 1;     /* written since read */
        } role[PROCESSES];
```

1.2.4. Compiling (or interpreting) CM Code

Two libraries are necessary for using common memory. cmlib.a is the common memory client code. This uses a lower-level library, libstream.a, which provides connection and packet service on top of TCP. Both libraries normally live in /usr/local/lib.

Thus, to link common memory programs:

```
        cc foo.c -lcm -lstream
```

Include files reside in /usr/local/include/cm. Normally, it is necessary only to include /usr/include/sys/time.h and /usr/local/include/cm/cm_user.h as follows:

```
        #include <sys/time.h>
        #include <cm/cm_user.h>
```

.lisp files are in /usr/local/lisp. There is one file provided to initialize common memory from lisp, cm_user.lisp. Thus to use common memory, you should execute the following:

```
        (load 'cm/cm_user.lisp)
```

1.2.5. Starting The Common Memory Manager

In the AMRF, the common memory manager executable program is located in directory /usr/local/bin. The common memory manager should be started before any processes are started. Anyone can start the common memory manager (i.e. you do not have to be root or have the same uid as any other users of the common memory manager). Just type:

```
        /usr/local/bin/cmm
```

Normally, nothing else is required, however the common memory manager can take some arguments to modify the default behavior.

ARGUMENT -d [0-9]

This argument will cause the common memory manager to print out debugging information. Higher numbers evoke greater amounts of output, since the debugging information displayed includes all output generated by the lower numbers. Current debug level assignments are:

0	means no debugging
3	refers to message handling
6	refers to slot (message contents) handling
9	refers to most buffer/byte/string coping.

ARGUMENT -t seconds

This argument blocks waiting for up to this time period, if the client's kernel queue is full while the common memory manager is trying to send a message to the client. After the time out expires, the common memory manager goes on and will retry later. This typically implies that the client is hung, but is not always the case. The default time out period is 5 seconds.

ARGUMENT -p port_number

The initial connection port that the common memory uses may be changed by specifying the port number. The default is 1525. This is useful if you want to have multiple separate common memories on a single computer.

The common memory manager does not require a controlling terminal to run. Note, that if the common memory manager is killed, all the processes using it will terminate if they are writing at the moment that the common memory manager is killed. This is due to a SIGPIPE being sent to each of the clients. If you do not want this behavior, you should surround your calls to cm_sync with a setjmp/longjmp alarm, just the way one normally does with blocking writes. (The common memory manager does this internally to protect itself from the client's dying. You can look at it in man.c.) In typical use, however, people do not do this, since the common memory never dies of its own accord.

Once the common memory manager is running, you can start the user processes. Specifically, the common memory manager process should be started before the first cm_process_name() is executed. If a common memory manager process is started while one is already active, the second manager process will detect the presence of the first one, display a message reporting this fact, and exit.

1.2.6. Additional Lisp Notes

In Lisp, all functions are identical to their C counterparts. Common sense dictates usage differences. A small Rosetta stone is presented:

```
   For C:
        date = cm_declare(......);
        foo = cm_declare(.....,CM_ROLE_READER,CM_TYPE_LIST,
                               CM_PERIOD_FOREVER);
        cm_sync(WAIT);
        cm_set_value(date,"12 Dec...");
        cm_get_value(foo,foolist);

   For Lisp:
        (setq date (cm_declare 'date CM_ROLE_XWRITER
                    CM_TYPE_STRING CM_PERIOD_FOREVER))
        (setq foo (cm_declare 'foo CM_ROLE_READER CM_TYPE_LIST
                    CM_PERIOD_FOREVER))
        (cm_sync WAIT)
        (cm_set_value date "12 Dec...")
        (setq foo (cm_get_value foolist))
```

Note that uppercase values denote constants that should be evaluated before use (i.e. unquoted). For example, to check if cm_declare() returns without failure the code would look like:

```
        (cond ((eq CM_BAD_OBJECT (cm_declare ....)) nil)
              (t t))
```

1.2.7. Using Sun Common Memory With VAX Common Memory

The following section is only appropriate to people using the VAX's HCSE code under VMS.

It is possible to run the same code on the Sun (Sun CM) and the VAX (HCSE CM) if certain steps are taken.

An interface library is supported that effectively replaces the Sun CM user calls with calls into the HCSE library. This library is currently available in ~libes/cm/src.5v.

Code using the UNIX CM library with HCSE should add the following parameters to the link command (in a .opt file)

```
        user1:[libes.cm.src$5n5v]suncmlib/lib,
        cm_library:cmlib/lib,
        psect=cm_shrmem,page
```

Versions of the code compiled for debugging are available by specifying:

```
        user1:[libes.cm.src$5n5v]suncmlib_debug/lib,
        cm_library:cmlib_debug/lib,
        psect=cm_shrmem,page
```

1.2.7.1. Restrictions

1.2.7.1.1. Types

The type systems in both the Sun system and the HCSE system are quite different. The HCSE types are based on Praxis. Primarily this means that they have user-definable types. Secondly, there are no arbitrarily-sized data objects.

Two extra parameters on the cm_declare statement exist in the Sun CM to get around this. The first is a maximum size. The second is a pointer to a type structure. If the type structure pointer is zero, the size is used to automatically select a type structure. For more information about creating type structures, see the HCSE programmer's reference manual [15].

A typical call that would be portable to both the Sun and VAX systems looks like the following:

```
        if (!(date = cm_declare("data",CM_TYPE_SIZED,
              CM_ROLE_XWRITER, CM_PERIOD_FOREVER
#ifdef VAX11C
              ,1000,0
#endif
                       ))) {
              printf("declare of var failed\n");
              exit(-1);
         }
```

When compiled by the DEC C compiler, the additional two parameters will be included.

1.2.7.1.2. No Queued Updates

The HCSE CM does not support queuing of variable updates. This means that if one process goes to sleep while a second process writes a variable several times, if the first process then wakes up, it will see only the last values written by the second process, not all the intermediate ones.

A different explanation of this phenomena is that there is no difference between specifying CM_WAIT_AT_MOST_ONCE and CM_WAIT_FOR_ALL in cm_sync().

1.2.7.1.3. No Command Associations

The third difference is that the HCSE does not store command associations with variables in the common memory manager itself. (In fact, they are just dropped on the floor). Therefore, routines like cm_status_equal() don't work.

If you need command associations or their effect (and most people do), you must stuff in a number in front of every variable indicating how many times this variable has been written. Then create a new variable that holds the old value of the number which you can use to compare it against.

A package that implements this along with current mailbox types in use in the AMRF lives in k:~network/mailbox. An example using these functions is in the cm source directory in vws.c. vws.lisp is a Lisp version of the same thing.

1.2.8. Unusual Types

Most types have the obvious meanings. Unusual types will be discussed here. The following is planned but unimplemented.

CM_TYPE_SEMAPHORE is a type for performing synchronization. The only operations defined on a semaphore variable are wait() (p) and signal() (v). They have the standard meanings as shown below. Note that use of either wait() or signal() forces common memory to be accessed (on the spot).

```
cm_wait(sem)
{
        wait until sem>0;
        s--;
}

cm_signal(sem)
{
        s++;
}
```

1.2.9. Basic Limitations of This CM Implementation

Certain limitations exist in the common memory manager. It is possible to change any of these and recompile. Changeable limitations (and their defaults as the system is distributed are):

```
/* cm.h */
CM_MSGSIZE       100000  /* Maximum size of any single set of
                            variable updates between the
                            common memory manager and a user */
CM_SLOTSIZE       20000  /* Maximum size of any single variable */
PROCESSNAMELENGTH    20  /* Maximum length of the process name */
VARIABLENAMELENGTH   20  /* Maximum length of any variable name */


/* cm_man.h */
CM_MAXVARIABLES      50  /* Maximum number of variables local to
                            the common memory manager */
CM_MAXPROCESSES      20  /* Maximum number of processes that can
                            to the common memory manager
                            simultaneously.
                            Absolute maximum of 32 (or number of
                            user file descriptors) under 4.2BSD.*/

/* cm_user.h */
#define   MAXUSERVARIABLES 100   /* Max number of variables local
                                    to a user */
```

1.3. Multibus Implementation

The Multibus common memory interface is tightly coupled with the NIP software, and is presented in Section V.2.1.3., below.

For descriptions of the user interface, the interested reader is referred to the appropriate Robot Control System documentation.

1.4. Hewlett-Packard 9000 Series 200 Implementation

The NIP interface to common memory is documented in the NIP reference section. For descriptions of the user interface, the interested reader is referred to the appropriate Inspection Workstation documentation.

2. NETWORK INTERFACE PROCESS (NIP)

2.1. Serial Asynchronous NIPs

The following subsections describe the various asynchronous NIPS.

2.1.1. DEC VAX 785 Implementation

The VAX NIP is virtually identical to the Multibus implementation, but takes advantage of a significant number of system calls available through VMS to provide timer, dispatcher and asynchronous trap (AST) service. The interested reader is referred to the VAX NIP listing for examples of the incorporated VMS system calls, and to the Multibus documentation for program module documentation.

2.1.2. Sun Microsystems Implementation

The Sun NIP is a port of the 68000 (Multibus) NIP. The following subsections identify different problems that had to be overcome in developing the Sun implementation from the Multibus NIP.

2.1.2.1. Implementation Approaches

There were two approaches possible for the implementation of the Sun NIP:

(1) do what we had done previously, and install a board with the NIP software into the UNIX computer systems, and

(2) port the NIP software into UNIX directly.

Approach (1) looked more difficult for several reasons:

(a) Our UNIX systems came in all shapes and sizes, and we lacked the support to keep up with the rapid influx of new hardware. Software support seemed easier.

(b) Buying boards for the numerous systems was expensive, whereas UNIX processes are free.

(c) We estimated that the software effort to port the NIP as user code was less than writing a special driver to support the NIP on a board.

Having decided to implement the NIP directly into the UNIX environment, several recent additions to 4.2 BSD UNIX were critical:

(a) software interval timer with accuracy of milliseconds,

(b) asynchronous input/output, and

(c) software signal facilities.

2.1.2.2. Shared Data Structures

Several data structures were shared between the cooperating processes of the NIP. While 4.2 BSD promised shared memory, it failed to deliver in this respect, so we decided to try simulating multiple processes in a single UNIX process. This would give us the ability to have shared data structures: something which is essential to the original design of the NIP.

In order to accomplish this, we retained the original overall design of the NIP as a minimal operating system. We included the original scheduler and dispatcher (with some minor modifications) from the stand-alone NIP, along with the original processes as subroutines.

The main route was then rewritten to look like

```
main ()  {
     fg(process_1);
     bg(process_2,10);
     bg(process_3,17);
     .
     .
     .
     dispatch();
     }
```

The second argument of bg (background) specified that this process (subroutine) was to be run after that many time units (milliseconds, here).

Processes could start up other processes after the dispatcher had started, if desired.

While the subroutine/processes were viewed as co-routines, the only way to return control to the dispatcher was to explicitly return(). All background tasks were expected to return at a convenient but brief interval of execution time. The tasks were then rescheduled for execution. During re-execution, each task started by examining a state vector and continued processing from where it had stopped earlier.

A more sophisticated solution would have been to provide separate execution stacks for each subroutine started by bg(). Of course, this leads into the complex topic of scheduling. Auxiliary arguments to bg(), such as maximum time quantum, priority, etc., would be useful. Our solution took the easy way out, at a small expense in user coding effort.

2.1.2.3. Timers

The stand-alone NIP made use of a MC6840 timer chip to generate timer interrupts at a known interval (10msec). At each interrupt, a queue was examined for expired timers.

We knew 4.2 BSD supported subsecond interval timings, however we were unsure how much of a load constant 10 msec interrupts would place on the system. Here, each interrupt was to be processed by a user interrupt handler, meaning that at least one context swap would occur at each interrupt. To reduce this interrupt overhead, we decided to continually schedule the next interrupt for the next shortest timer (which was presumably longer than 10 msec).

This made the timer code slightly more complex than the original. For example, when adding a timer the original code simply added a timer entry. The 4.2 BSD code had to check first if the new time out was shorter than the timer currently being waited for. If so, the time out was cancelled and rescheduled.

Because the timer was not scheduled at a regular interval, we expected a skewing of time, as our own code to handle the timers after they have occurred (or were cancelled) but before the next one is scheduled takes time to execute. Since accurately predicting measurements of user code in real-time is impossible in UNIX, this is corrected by occasional comparisons against the time-of-day clock.

2.1.2.4. Input/Output

Because the Sun NIP ran on top of UNIX, there was no need to provide device drivers at the physical layer. All access to devices went through the UNIX device drivers. This had both advantages and disadvantages.

The obvious advantages were that we did not have to worry about supplying devices and their low-level drivers. All UNIX devices were accessible at the system call level. This immediately leads to the primary disadvantages.

The I/O system used by the stand-alone NIP was heavily based on the VMS model. VMS provides very complex options for I/O, such as functions to be called at I/O completion (or timeout), not supported by typical UNIX drivers.

UNIX provides asynchronous I/O but with an extremely crude interface through fcntl, select and read/write. If I/O is attempted when such an operation would block, the system call returns EWOULDBLOCK. If partial reads or writes can be done, the number of bytes transferred is returned, and the I/O must be rescheduled again until completed or EWOULDBLOCK is returned.

In order to avoid blocking, a signal (SIGIO) can be requested to be delivered upon the ability to read or write without blocking. In order for this to be useful, I/O operations were attempted immediately and queued if not (or partially) successful. A SIGIO handler was written that called select() to determine where I/O was pending. By comparing it to the queue of user requests, it was able to match against I/O operations of interest, and return or resume the I/O request. At I/O completion, the user completion routine was called.

Timeouts were handled by declaring a background process (using bg) that would trigger once after the appropriate interval, deleting the I/O entry and calling the user's function that indicated the I/O had failed by timeout.

We used the serial ports on the Suns. However, their primary drawback was that the Sun drivers were quite limited in the amount of throughput. They were quite slow, and had limited input buffering capability (approximately 256 bytes). Receiving packets larger than the device driver's input buffer size or receiving them too quickly back-to-back would put the driver into a hung state.

2.1.3. Multibus Implementation

These sections describe the NIP, as written for implementation on the Omnibyte and Pacific Micro 68000 boards installed in the Multibus-based systems (all controllers of the Horizontal and Turning Workstations). Individual modules are identified and discussed.

The NIP is an event driven system. The general operation is:

(1) interrupt routines service devices and set event flags,

(2) asynchronous traps (ast) are triggered by the event flags but operate at a main processing level (not at an interrupt level) which do most of the data movement and protocol,

(3) background routines run the least time sensitive activities such as updating mailboxes and putting out the status of the NIP.

The software is written almost exclusively in 'C', with the exception of a startup routine (ISRVEC) and some system specific, defined variables (<station>DEF). The modules are broken up along OSI lines mostly, although Copymail performs both the application interface function and the session function, and the link modules have a conceptually messy interface.

Control travels through the protocol stack generally by direct calls down and ast calls up (the routines in the next higher level to be called by ast were specified by the higher level earlier). When appropriate, a status will be returned to the caller (when accessing a device or indexing a table), otherwise a desired value can be returned. It is also possible that nothing will be returned.

The code is completely contained within ROM on the boards. At startup the code and static tables are copied into RAM and the initialization sequence is executed.

2.1.3.1. COPYMAIL

COPYMAIL is the main process in the Network Interface Process. It establishes mailbox connections, services those connections (by checking for changes in outbound mailboxes and updating incoming ones), and executes commands and reports status through the NIP command mailboxes. This runs in the background processing mode.

The users on these systems see the network as memory areas that they read and write to. The memory areas are structures of variable length, the actual structure and protocol for use of which is listed in Appendix A of this document.

2.1.3.2. XPORT

XPORT (transport) is called by copymail to get guaranteed, error free communication service. This done by having all packets acknowledged by its peer xport entity. If after a significant time the packet remains unacknowledged then an error signal is sent up to copymail. Details of this protocol and structures used for it are in Appendix G.

2.1.3.3. NETWORK

This module provides routing services for the NIP. An identifier which is ROMed onboard and can be found in <station>def.a68 (see below) as _defnadd or is determined by querying the Ethernet board for its address, is the network address and is used to index into a table (NETGEN) for gaining information about the local node and possible routes to other nodes. Whenever a connection is desired, the goal site name is passed through to network and it is used to find an entry into the table in NETGEN which has the routing information. NETWORK then uses this information to build the 8473 network header and request the appropriate link service to send the packet to the next node in the route.

2.1.3.4. MLINK

MLINK provides link services using devices that can send to multiple nodes (ie. Ethernet). It accepts I/O requests from the network and queues them. It establishes a table of open links and keeps track of the links' status.

2.1.3.5. SLINK

The function of SLINK is similar to MLINK but for serial communications interfaces. However, since the serial lines do not have any implicit packetization method, packetization must be done here also. The packet format is derived from HDLC, but the control information is used only to determine the length of the packet and its validity.

2.1.3.6. IO

The IO module is an I/O dispatcher, it provides a single interface through which all the user services can access I/O channels. Through IO the user can: initialize a device (really meant to be used to reset a device that has gotten into a weird state); open a device for a particular function: read, write, both or neither; post a read,write or both; request an id (currently, this is an Ethernet address from an interface board); close a device.

The following is a description of the I/O calls that a user might want to know:

CALL _open(dev, func, chan)

PURPOSE to open a communications device for specified function

ARGS dev- a pointer to a two character ASCII word where the first character specifies the device type by an A-F and the second specifies the device number 0-F. Device type determines which entry into the table in CONFIG will be used to select driver routines to be called.

func- a constant which specifies how the device will be used: IO_W,IO_R,IO_RW and IO_NRW (used for ID) These can be found in IODEF.H68.

chan- is a pointer space where the routine will put a two byte signed integer channel number.

CALL _close(chan,dev)

PURPOSE to turn off an I/O device and free an I/O table entry

ARGS chan-the 16 bit integer value which had been passed to the user when he did an _open()

dev-a pointer to space for the routine to put the two character device specifier that was used to open the device.

RETN returns a 32 bit signed integer status which was returned from the driver or NOCHAN if the chan argument was bogus.

```
--------
CALL    _io(func, chan, buffer, length, option, status,
        ast, arg)

PURPOSE this is the tool with which the user can get access to an
        active channel.  The user posts a request for the desired
        action and _io returns a status as to the success of the
        request.  The action will be done by the driver through
        interrupt and ast level processing and the result of the
        action will be returned to the user through the ast he
        passed to _io().

ARGS    func-   is an action specified by a constant in
                IODEF; the possibilities are:
                     IO_R   read
                     IO_W   write
                     IO_ID  used to get net address from Excelan.

        chan-   16 bit int specifying channel number
        buffer- pointer to memory area to manipulate data in
        length- length of the message to be read or written
        option- variable; the interpretation of which
                depends on the device being used.  It is
                usually used as a timer length field.  When
                it is found not to be zero, the driver
                starts a timer for 'option' ticks after
                which the 'ast' will be triggered.
        status- a pointer to the status block to be used
                for this transaction
        ast-    routine to be called after transaction is
                completed by I/O device.
        arg-    argument to be used when calling the ast
                routine.

RETN    a 32 bit integer indicating the success of posting
        the I/O request. Valid values for this status are
        in IODEF.



--------
CALL    _inidev(dev)

PURPOSE this allows the user to reinitialize the device
        when it gets in a hung state.

ARG     dev-    two character ASCII string indicating
                device type and number.
```

2.1.3.7. EXCELAN

EXCELAN is the device driver for the Ethernet board. It exposes a packet oriented interface to its user. The user requests a service such as a read or a write with corresponding buffers and I/O status blocks (which include a transfer count and a status variable), the driver works with the number of bytes requested of it and returns the buffer and a status to the user.

2.1.3.8. ACIDVR

This provides much the same services as above, except that link packets have to be built by the user before being received by ACIDVR.

2.1.3.9. Special Modules

2.1.3.9.1. NIPMAIN

NIPMAIN performs the following functions:

(1) run the initialization routines for all modules,
(2) start the background process copymail()
(3) start the background process loader()
(4) start the background process tester()

2.1.3.9.2. CMBUFMGR

CMBUFMGR manages buffers which are used for actual data. While there are three types of buffers (small, normal and large) defined in cmbuf.h, only two types are managed: medium and large.

Small buffers are used for exchange of control information between peer protocol layers and are usually a part of the control block of the module for that layer.

The manager controls the repository of normal and large buffers. The user interacts with the buffer manager through the routines cm_getb, cm_putb, cm_waitb and the analogous large buffer routines cm_getl,cm_waitl.

Function descriptions:

```
--------
CALL    cm_getb(func)

PURPOSE to get a buffer from the manager

ARG     func = INPUT or OUTPUT, what the buffer is to be
        used for.  This is because buffers for input are
        given priority.  When the buffer pool is low, only
        input buffers will be issued.

RETN    a pointer to a normal buffer (type = cmbuf)



--------
CALL    cm_putb(buf)

PURPOSE to give a buffer back to the manager

ARG     buf is a pointer to the buffer to be returned

RETN    void



--------
CALL    cm_waitb(call, arg)

PURPOSE to wait for a buffer when there aren't any available

ARGS    call is a pointer to the routine to be called when
        there are buffers available

        arg is the argument to be passed to the routine call



--------
CALL    cm_getl(func)

PURPOSE to get a large buffer from the manager

ARG     func = INPUT or OUTPUT, what the buffer is to be
        used for

RETN    a pointer to a normal buffer (type = cmbuf)
```

CALL cm_waitl(call, arg)

PURPOSE to wait for a large buffer when there aren't any available

ARGS call is a pointer to the routine to be called when there are buffers available

arg is the argument to be passed to the routine call.

2.1.3.9.3. CONFIG

CONFIG has links to device driver routines corresponding to I/O channels.

2.1.3.9.4. DISPATCH

The dispatcher controls all non-interrupt level processes after nipmain() relinquishes control at start up. It does this by maintaining a list of processes to be executed. These processes come in two varieties: AST level and simple background process level.

The dispatcher checks for pending ASTs (set event flags) first. When it finds one it removes it from the table and excecutes it.

When finished with the ASTs, the dispatcher will move on to processes. If it finds one pending, it will execute it, then check for ASTs again. A process will stay in the table until the user removes it, and the dispatcher does not turn off the event flag (which shows that a process is pending).

Dispatch.c68 also contains routines for creating and canceling ASTs, and logging events into a trace area in memory.

CALL _dclast(uevt, uast, uarg)

PURPOSE put an ast entry into the process table

ARGS uevt- event flag to be checked by dispatcher which indicates that an AST is pending

uast- pointer to routine to be called when the dispatcher finds that the AST is pending

uarg- argument to be passed to the uast routine

RETN void

```
--------
CALL    _dclpro(uevt, upro, uarg)

PURPOSE to add a background process entry to the process list

ARGS    uevt-  pointer to the event flag which indicates to
               the dispatcher to call the process

        upro-  pointer to the process to be called

        uarg-  argument to be passed to the process


--------
CALL    _canast(uevt, urtn, uarg)

PURPOSE to remove an AST entry from the process table

ARGS    uevt-  pointer to event flag which is used to
               compare against table entries

        urtn-  routine that would have been called if the
               AST had ever been pending

        uarg-  argument that would have been passed to the
               AST routine had it ever been pending before being
               cancelled


--------
CALL    _disptr(uast)

PURPOSE to record the address of a routine in the trace
        area on board.  The trace area is a circular buffer
        therefore it will eventually overwrite earlier
        entries.

ARG     uast is the address of the routine the use of which
        is to be logged.  Note that one could put any 32
        bit number here.
```

2.1.3.9.5. ERROR

This module provides the routines necessary to open and use an RS-232 port for error reporting. The defined constant ERRDEV determines which device will be used for this purpose. At the moment it is hardcoded to use port 1 (this is the second port). It only supports strings and looks like this:

```
err_out("<error report>")
```

2.1.3.9.6. LOADER

This module is used to testing the network. Routines in this module create a mailbox with a nonsense but unique ASCII message and increment its sequence number. This makes it very easy to create a large load by connecting to this mailbox through whatever link one wants to load. All that has to be done to use this is to connect out to load_mbx. It is always active. There is another pre-established mailbox for testing purposes called test_mbx. The NIP doesn't do anything to it, but the space and name are reserved.

2.1.3.9.7. MBXIO

This module contains routines for reading and writing to mailboxes.

2.1.3.9.8. SFUNCS

SFUNCS is a collection of very simple utilities which are used for string manipulation and word order conversion.

```
---------
CALL     bcpy (s, d, n)

PURPOSE copies n bytes from s to d

ARGS     s-   a pointer to the start of the source array
         d-   a pointer to the start of the destination array
         n-   number of bytes to be copied



---------
CALL     scpy (s, d)

PURPOSE copies bytes from s to d until a 0 is copied

ARGS     s-   pointer to 0 ended source string
         d-   pointer to beginning of area to put copied string



---------
CALL     bcmp (a, b, n)

PURPOSE compares two n-byte strings, a and b, and returns
         -1 if a<b, 0 if a=b, +1 if a>b

ARGS     a-  pointer to first byte of first string to be compared
         b-  pointer to first byte of second string to be compared
         n-  number of characters of the strings to be compared
```

```
--------
CALL    scmp (a, b)

PURPOSE compares two 0-terminated strings, a and b, and
        returns -1 if a<b, 0 if a=b, +1 if a>b
        The shorter string is extended on the right by 0s.

ARGS    a-  pointer to first byte of string to be compared
        b-  pointer to first byte of second string to be compared



--------
CALL    cvtup(s)

PURPOSE converts a lower case ASCII character to upper.

ARG     s-  a pointer to an ASCII character



--------
CALL    slen(s)

PURPOSE finds length of a 0-terminated string pointed to by S

ARG     s-  a pointer to a 0-terminated string



--------
CALL    cpswp(a,b,lb)

PURPOSE copies 16-bit word from a to b swapping bytes

ARGS    a-  pointer to word to be copied
        b-  pointer to place for copying swapped word



--------
CALL    wcpy(s,d,n)

PURPOSE copy n bytes as words from s to d

ARGS    s-  pointer to first word to be copied
        d-  pointer to place to copy words to
        n-  number of words to be copied
```

```
--------
CALL     word bswap(w)

PURPOSE swap the two bytes of the word at W and return word

ARG      w pointer to the beginning of a word

RETN     byte swapped words


--------
CALL     int4 lswap(lw)

PURPOSE invert the four bytes of the longword at lw and  return
         the word

ARG      lw-  pointer to the beginning of long word to be swapped

RETN     swapped long word
```

2.1.3.9.9. SYSINI

SYSINI is called from isrvec to initialize the dispatcher, the clock and io(the system) and start the dispatcher.

2.1.3.9.10. TESTER

TESTER is a null function slot for adding optional test routines.

2.1.3.9.11. TIMER

This module incorporates both the timer device driver and the routines for providing the user with time oriented functions such as setting an event flag after a specified number of ticks or calling a routine (at the highest level of maskable interrupt) after a specified number of ticks.

```
--------
CALL     _setimr(intval, event, ast, arg)

PURPOSE  put an entry into the timer list and activate an AST
         after the specified number of ticks

ARGS     intval- number of ticks to wait before setting the
                 event flag on the AST

         event-  pointer to event flag to be set when
                 interval has expired

         ast-    the routine to be called at end of timer

         arg-    argument to be passed to the ast routine
```

2.1.3.10. Assembly Language Components

2.1.3.10.1. *DEF Files

These files are actually of the form <station>def.a68 where <station> is a node name such as HWS. This is where all the station specific configuration information is supposed to be stored. Typically, it will contain the Multibus I/O address of off-board I/O devices, number of I/O device types (for instance, two for a station with a serial port and an Ethernet board), external ram access address of onboard memory, station id, and a network address. This module is part of the static tables copied into RAM with the code.

For example, the following code is the station specific definition file for the Horizontal Workstation Controller (HWSDEF.a68):

```
__exclpor:* I/O port address for Excelan
        .long   0xff00d0        * Ethernet board, 32 bit integer

__iomaxd:* number of device types on this
        .long   0x000002        * system

__ramoff:                       * the RAM starting address on
        .long   0x300000        * this board as seen over the bus

__station:                      * station identifier
        "HWS"
        .byte      0

_def_nadd:                      * station network address
        .byte   0x08            * (also Ethernet address)
        .byte      0
        .byte   0x14
        .byte      0
        .byte   0x39
        .byte   0x66
```

2.1.3.10.2. ISRVEC

This module does the preliminary set up on board at reset: it fills the interrupt vector entries, sets the stack pointer and sets up for calling "C" routines. There is also a routine in this file to set the interrupt mask.

```
--------
CALL    spl(levelSR)

PURPOSE to set the interrupt mask level

ARG     value the user wants to set the 68000's status
        register to. The format for this is 2X00 where x
        is the value of the interrupt priority.

RETN    the value the status register had before calling spl
```

2.1.4. Hewlett-Packard 9000 Series 200 Implementation

The IWS version of the AMRF network software was created primarily by translating the existing C version for the 68k Multibus systems into HP Pascal. Both languages are block structured with many of the same control features, which made such a direct translation possible.

2.1.4.1. HP Pascal Extensions and Imported System Modules

2.1.4.1.1. Imported System Modules

$SYSPROG$
: system programming language extensions such as TRY...RECOVER, anyvar, anyptr, sizeof, etc.

$USCD$
: ucsd version of pascal extensions

import sysglobal, sysdevs
: enables direct programming of system devices such as internal clock, timers and keyboard

import iocomasm
: provided bit manipulation functions such as BINIOR, BINAND, BIT_SET, etc.

2.1.4.1.2. Pointer Variables

In the rush to complete the translation/development, things such as Pascal's "pass by reference", etc., were not used everywhere they could have been. Instead, the HP extension of pointers was employed to support direct translation from C pointer variables to HP Pascal pointers.

2.1.4.1.3. Translating The C "RETURN" Statement

Pascal has no equivalent of the "return" statement in C. Indeed there is no need for such a statement, as one can always restructure code to avoid its use. In translating the use of "return" statements to Pascal, we made use of the Pascal "goto" statement instead of restructuring the code. "Goto 999" is used wherever C code had used "return" as a premature exit from a procedure. Label 999 is typically at the end of the procedure. "Goto" was used for several reasons: 1) sometimes restructuring code to not use "return" makes it much more difficult to read and follow , and 2) we were in a big hurry to get the IWS software running, and keeping the Pascal code looking similar to the original C code speeded up the debugging process.

In retrospect, the ucsd extension "exit" could have been used to replace the "return" statements quite nicely.

2.1.4.2. Pascal Strings Versus C Strings

Pascal stores strings as a byte count character (0-255) followed by that number of characters. All of Pascal's inherent string functions depend on this string structure. C stores strings as a null-terminated string (i.e., the string followed by a NULL character). Strings are handled in the IWS network code internally as Pascal-type strings. However there are a few instances where communications with other network systems in the AMRF require the use of C-style strings. One case is the sending of string information to and receiving of string information from "netcmd". In these cases, special functions were included in the sfuncs module to translate between the two string types.

2.1.4.3. Pascal Type Checking And Intermodule Dependency

Pascal is more rigorous than C in checking data types, function/procedure usage ,etc. For this reason, the IWS network software adds many new data type definitions that are not in the C version. These are data types that are not in the C version because C permits one to do dangerous things like mix pointer types, usage of function return values, etc.

In addition, module dependency relationships are required to be more explicitly defined in Pascal than in C. In several cases, modules had to be modified because a straight translation of the C code implied that two or more modules depended upon each other or otherwise created a "dependency loop". For example, module A depends on module B, module B depends on module C, and module C depends on module A. This type of situation is not directly supported in HP Pascal. In these cases, one direction of dependency is eliminated by passing the required information at run time instead of link time. For example, xport calls "net_sethooks" at run time to tell the network layer the procedures to call when it has information to send to the transport layer, i.e. the upwards "hooks" into the transport layer. This way transport could depend on network layer, but there was no longer a need for network to depend on transport layer.

2.1.4.4. Debug Variables Added

Debug variables were added to each module to allow the user to track the network operation via print statement to the terminal screen. Each module has a debug variable that can be set by the operator at startup time. Default is zero. Non-zero values trigger print statements.

2.1.4.5. Interaction With The IWS Common Memory System

The network software's main responsibility is to pass common memory variables between the local common memory system and the network. All common memory variable requirements are known ahead of time (before starting up the IWS). Therefore all variables are declared at startup time by the user program, and not dynamically at run time by the network software as in other systems such as the VAX and SUNs. When copymail receives a command to connect a common memory variable, it merely checks the common memory system (CMS) to make sure that the variable exists and corresponds to the desired attributes of name, size, direction of transfer, source/destination sites, etc. If the check is unsuccessful, a message is printed on the console.

In addition to the standard INPUT and OUTPUT mailbox types, a new type called "PASSTHRU" was invented. This was required due to the configuration of the IWS network (IWS as the central hub in a star configuration with CMM, SRI and RCS at the end of each link). All information to/from each of the auxiliary stations had to go through the hub IWS station. Since the existing network layer software has no provision for relaying messages at the internet layer, it was decided to do the message relaying at the application layer. PASSTHRU mailboxes only exist on the IWS system. Their special characteristic is that they may be connected both as INPUT and OUTPUT, and may be connected to multiple remote AMRF sites. Although this was previously possible to do with the network software, the IWS CMS had no previous provisions for multiple directions for one common memory variable.

2.1.4.6. User Control During Time-Critical Periods

Subroutine extensions were added to the network software to allow the IWS controlling software to manipulate the allocation of CPU time to network software during certain critical time periods. These routines are:

```
enable_nip()/disable_nip()      and
enable_clock()/disable_clock()
```

2.1.4.7. List Of Modules

Following is a list of the NIP modules on the IWS with some commentary on some of them. Basically they fall into three categories:

(a) those that are straight translations (with maybe some additions) from the corresponding C version,

(b) those that differ radically from the C version, and

(c) those that were not in the C version

2.1.4.7.1. Modules that are Straight Translations

bufmgr.tex
cmbuf_h.tex
cmlkst_h.tex
config.tex
copymail.tex - modified to include the optional generation of mailboxes locally via the netcmd module. The operator of the IWS workstation controller has the option of generating the mailboxes locally via script files driving the netcmd module or by accepting commands from the network manager running on the remote host "VAX"

Checks made on incoming mailbox connect commands:

1) that it is common memory
2) that the size request matches that of the CM variable
3) that the direction of transfer matches that of the CM variable. In the case of the new PASS_THRU variable type, either INPUT or OUTPUT would pass this test.

The Boolean variables copy_done and copy_enabled were created and used to enable higher level software (ECS) to lock out network access to common memory during critical times or to speed up processing during critical time periods such as robot movement.

dispatch.tex - The procedure nullproc() was added since null procedure pointers could not be sent as arguments. Dispatch is run from the ECS kernel. Copymail is a process run every time through the dispatcher. An event flag mask was added beyond the C version to accommodate problems with reading the dual-port ram on the Async I/O card.

errors_h.tex

io.tex

iodef_h.tex

mbx_h.tex

mbxio.tex - read/write locks taken out of mailbox i/o since there is only one processor and all i/o is completed before leaving the task. The "one processor" attribute is expected to remain since HP has no multiprocessor version of the same computer and there is no expectation to insert a specialized NET processor into the HP bus.

mbxmgr_h.tex

net.tex - net_sethooks() created so transport layer module could import network module and still be able to dynamically (run time) set the hooks (i.e. addresses of procedures) for network to call for the cases of sending "signals" to transport and queuing the input packets up to transport layer. net_sethooks() is called at initialization time by transport (from xp_ini). This way, network sets up its hooks to transport only once at startup and the hooks to transport are valid for all network connections.

net_ini() in turn calls sl_sethooks() so that the link layer can similarly set its upward hooks to network layer for signals and input queuing.

net_h.tex

netgen.tex - we added the site identification code here. This code identifies the IWS station (IWS, CMM, IRC, SRI) by its switch value on the I/O card

nipini.tex - we added the initialization of debug variables here. There is a debug variable for each module in the network software. The operator is asked at startup time to set any desired non-default values. Default value for all debug variables is zero.

params_h.tex

sfuncs.tex - Extensions were added here beyond the C version.

copy_name_to_str and copy_str_to_name convert between C-style null-terminated strings and Pascal-style byte counted strings. This was necessary since all internal string processing was done using the native Pascal strings and string operations, but there remained a need to communicate string information to/from (among others) netcmd which understood only null-terminated strings.

```
        Function gettoken was added for use in parsing
        user-entered ASCII strings.  Used in netcmd and nipini.
slink.tex
timer_h.tex
types_h.tex
xport.tex
        xp_ini calls net_sethooks procedure so network layer can
        set it's procedure variables for procedures to call for
            a) queuing input packets to the transport layer and
            b) sending signals to the transport layer
        otherwise xport is a faithful translation of the C code
```

2.1.4.7.2. Modules that Differ Radically

acidvr.tex - driver for the 98691 asynchronous smart i/o card. Written in the same format as other C version drivers, but obviously had to be written hardware dependent for the HP microcomputers. Driver also includes the Z80 assembly language interface software written to execute on-board the I/O card.

timer.tex - also strongly resembles the C version with many of the same calls performed as direct translations (canast dclast setimr, etc). However, the actual programming of the HP's hardware clock and handling of interrupts had to be changed for this hardware.

We added clkclose procedure which merely stops the clock hardware from interrupting and restores the system's original clock handling routine as the clock interrupt vector. This is to be called when the ECS exits and returns control to the console.

We added enable_clock and disable_clock procedures. These temporarily enable and disable the hardware interrupts of the clock and are called to lock out the interrupts during certain critical periods of real-time operation by the robot controller and the SRI system.

2.1.4.7.3. New Modules

netcmd.tex - This code is a translation of a subset of the netcmd software from the VAX. It enables the creation of mailboxes locally without the aid of netcmd running on another system.

Copymail was slightly modified to accommodate this feature. Script file structure is identical as for the VAX network manager.

station_h.tex - include file used by netcmd.

2.2. Ethernet (TCP/IP) NIPS

2.2.1. DEC VAX 785 Implementation

2.2.1.1. Introduction

The VAX TCP Interface to the AMRF network is a set of routines written in the C Language which allow the calling program to connect, disconnect, read, write and get status via the TCP transport protocol over the ethernet.

These routines are used by the AMRF Emulation Program, Copymail, which was originally written for the AMRF Serial Interface. In order to incorporate the TCP Routines into Copymail, the following changes to Copymail were made:

(1) All XP_ calls were replaced by TP_ calls.
(2) The SERVICE routine was rewritten.
(3) The return from a connection request was rewritten.
(4) Output is blocked unless fully connected.
(5) A debug option (d) at execution time was added.
(6) A port number option (t) at execution time was added.
(7) A bug was fixed in the connection code.
(8) A number of other small changes were made.

The resultant program is called TCPNIP and resides with the TCP Routines, TCPLIB.C, in USER1:[NETWORK.TVAXV1]

In order to communicate through the TCP Transport layer these routines open two kinds of sockets:

(1) SERVER - A public socket whose port number is available to all TCPNIP's. Used to initiate connections.

(2) CLIENT - A private socket assigned to a each particular client (or site) when a connection is accepted.

All data transmission takes place through the client sockets.

Modules called by the main application program, TCPNIP, begin with TP_ and modules called internal to the TCP routines begin with TCP_. Communication from TCPNIP to the TCP routines are through the TP_ function calls. These calls are used to request services from the TCP routines. Control is returned immediately to TCPNIP and the services are performed asynchronously. Communications from the TCP routines to the TCPNIP are through the SERVICE module located in TCPNIP and specified with the connection request (ssap argument).

Communication with the TCP Transport Layer is done through VMS communication channels using VMS system calls SYS$QIO and SYS$QIOW. The channels are obtained and released using the VMS system calls SYS$ASSIGN and SYS$DASSGN. Reception and transmission as well as connection requests are done asynchronously using SYS$QIO. Other services such as creating a socket and shutting down a socket are done synchronously using SYS$QIOW.

NETGEN is a module which generates the network site names and addresses. This step should not be necessary since there are standard TCP routines from the vendor to supply all network addressing and site naming information. However, these routines have not worked in non-UNIX environments.

2.2.1.2. TCP Control Table

The TCP routines must keep a record of each open connection. The status of the open connections are kept in a TCP control table with one entry for each open connection. Each entry in the TCP control table will be referred to as a TCP control block or TCPCB.

A TCPCB is initiated when a connection is requested either by the local TCPNIP or the remote TCPNIP. A TCPCB is removed upon a disconnect request from the local TCPNIP or a disconnect request from the remote TCPNIP when the local TCPNIP never requested the connection. A TCPCB is retained upon a disconnect from the remote host if the local TCPNIP is connected. The TCP control table entry is defined by the structure TCPCB:

[SITENAME]	Records the name of the remote site.
[CONN]	Records the connection number (CID) (the index into the TCP control table entries + 1). A connection number of zero indicates an empty or deallocated TCPCB and a non-zero connection number indicates an allocated TCPCB.
[OUTQ]	Output queue holds the messages waiting to be transmitted to the remote site.
[INCNT]	Keeps count of the number of messages received.
[OUTCNT]	Keeps count of the number of messages sent.
[STATE]	Records the connected, half connected or shutting down condition. Equals zero or disconnected when the TCPCB is deallocated.
[INSIZE]	Records the number of character received during a reception.
[OUTSIZE]	Records the number of character transmitted during a transmission.

[CHAN]	Records the number of the I/O channel bound to the client socket. All I/O is done through this channel.
[RIOSB]	Used to return status and length information when input completes.
[WIOSB]	Used to return status and length information when output completes.
[SSAP]	Points to the module in the calling program which is executed when an event such as connection or input or output completion occurs.
[REMOTEADDR]	Records the internet address of the remote site. It is obtained from the tables built by NETGEN.
[IB]	Points to a communication buffer where the incoming data and the communication information is to be written. If no I/O is active, the pointer is null.
[OB]	Points to a communication buffer where the outgoing data and the communication information is found. If no I/O is active the pointer is null.

2.2.1.3. State Of A Connection

The STATE item in the TCP control table specifies whether the site is fully connected, half connected, disconnected or shutting down. Table V-1 lists the possible STATEs and the logical names that are used in the program and in this description of the programs.

Table V-1. VAX TCP NIP Connection States

LOGICAL NAME	TCP STATE	DESCRIPTION
TCPDISC	Disconnected	No entry in TCP control table.
TCPCONN	Connected	Both hosts actively connected.
TCPHLOC	Half Connected Local	Only local host has requested a connection or remote host has disconnected from a connected state.
TCPHREM	Half Connected Remote	Only remote host has requested a connection.
TCPSHUT	Shutting Down	Local host has requested a disconnect while I/O is active.

The STATE of an TCPCB changes upon a connect or disconnect request and in one case upon I/O completion. This case occurs when a disconnect has been requested for a site with output queued or input active (TCPSHUT). When the last I/O completes, the state changes to disconnected (TCPDISC). Table V-2 lists the events that cause a change in STATE.

Table V-2. VAX TCP NIP: Events Causing A Change Of State

EVENT	DESCRIPTION
LOCAL CONNECT	Local host requests a connection.
REMOTE CONNECT	Remote host requests a connection.
LOCAL DISCONNECT	Local host requests a disconnection.
REMOTE DISCONNECT	I/O error.
READ COMPLETION	Input completion while shutting down.
WRITE COMPLETION	Final output completion while shutting down.

Table V-3 lists the changes in STATE as a function of EVENTS. SAME indicates no change in STATE and NA (Not Applicable) indicates a condition that never occurs. Note: The change in STATE may depend on whether there is queued output (Q).

Table V-3. VAX TCP NIP: Change Of State As A Function Of Events.

INITIAL STATE	FINAL STATE					
	LOCAL CONN	REMOTE CONN	LOCAL DISC	REMOTE DISC	READ COMPL	WRITE COMPL
TCPDISC	TCPHLOC	TCPHREM	NA	NA	NA	NA
TCPCONN(Q)	SAME	SAME	TCPSHUT	TCPHLOC	SAME	SAME
TCPCONN	SAME	SAME	TCPDISC	TCPHLOC	SAME	SAME
TCPHLOC	SAME	TCPCONN	TCPDISC	SAME	NA	NA
TCPHREM	TCPCONN	SAME	TCPDISC	TCPDISC	SAME	NA
TCPSHUT	TCPCONN	TCPHREM	SAME	TCPDISC	TCPDISC	TCPDISC
TCPSHUT(Q)	TCPCONN	TCPHREM	SAME	TCPDISC	SAME	SAME

2.2.1.4. TCP Modules Called From TCPNIP

These modules are used to request services from the TCP routines. Control is returned immediately and the services are performed asynchronously. Communication from the TCP routines to the TCPNIP are through the SERVICE routine specified with the connection request (ssap argument). These six routines are called to either start the server program (TP_START), request a connection (TP_CONN) or a disconnect (TP_DISC), queue messages for transmission (TP_OUT), get status (TP_STAT) or get connection number (TP_ID).

CALL TP_START(ssap,port)

ARGS ssap - in - pointer to a module: specifies the default module to be called when an unrequested connection or unrequested input is received.
port - in - integer: specifies the server port number.

FUNC Generates the network tables and opens a server socket through which other sites can initiate a connection.

Calls NETGEN to generate all network site names and addresses.

Records a default SERVICE routine to be called when a remote site requests a connection before TCPNIP requests a connection (TCPHREM).

Records the local port number which was obtained either by a preset default port number agreed upon by all TCPNIPs or a port number set using the "t" option when the calling program was initiated.

Calls TCP_BIND to create a socket and bind it to the requested port.

If the BIND was successful, calls TCP_LISTEN to listen for a connection request from a remote site. The listen is asynchronous I/O and returns control to the program before any connection requests arrive.

Returns status of the LISTEN I/O request or the status of the BIND request if the BIND failed.

RETN Status returned by either TCP_BIND or TCP_LISTEN.

REFS NETGEN, TCP_BIND, and TCP_LISTEN

CALL TP_CONN(ssap, site, conn)

ARGS ssap - in - pointer to a module: specifies the module located in the calling program to be called when a noteworthy event occurs such as input completion or a connect or disconnect.

site - in - pointer to a character: specifies the name of the site to which the connection is requested.

conn - out - connection number (Connno): returns the connection number (CID).

FUNC Requests a connection to the specified site.

Checks to see if the site is already known (TCP_FIND_SITE) i.e. already has a TCPCB.

If fully connected (TCPCONN) or half connected locally (TCPHLOC), stores the connection number (CID) into argument conn, and returns the current STATE.

Allocates a TCPCB (TCP_GET_ENT) if this is a new connection.

Obtains the remote address (TCP_GET_ADDR).

Records SERVICE routine, connection number (CID) and remote address. Stores the connection number (CID) into argument conn.

Returns fully connected (TCPCONN) if connection with remote site has already been made (TCPHREM).

Calls TCP_CONNECT to make new connection. The connection is made asynchronously, i.e. returns immediately, before the connection completes.

Returns status returned by TCP_CONNECT, either half connected (TCPHLOC) STATE or connection failed (TCPFAIL).

At a later time when the connection completes, the calling program is informed via the SERVICE routine (ssap argument). If the connection is refused, the STATE remains in half connected locally (TCPHLOC) and remains there until the remote site connects at which time the calling program is informed via the SERVICE routine.

RETN STATE of connection or TCPFAIL.

REFS TCP_FIND_SITE, TCP_GET_ADDR, TCP_CONNECT

CALL TP_OUT(buffer)

ARGS <u>buffer</u> - in - pointer to a communication buffer (cmbuf): specifies the buffer containing the communication information as well as the text of the output message

FUNC Queues a communication buffer for output.

Checks the validity of the requested site, returns NO if not valid.

Adds the address of the communication buffer to the output queue for the CID requested in the communication buffer.

Initiate output (TCP_WRITE) and returns YES.

RETN YES or NO.

REFS TCP_WRITE

CALL TP_STAT(conn, buffer)

ARGS <u>conn</u> - in - connection number (Connno): specifies the connection number (CID) of the site for which status is requested.

<u>buffer</u> - in - pointer to a status buffer (cmlkst): specifies the buffer to receive the status information.

FUNC Supplies the status of the requested site.

Checks the validity of the requested site, returns NO if invalid.

Copies site name, STATE, input count, output count and flags into status buffer. Copies zeros if TCPCB is deallocated.

Returns YES if the TCPCB is allocated and NO if the TCPCB is deallocated.

RETN YES or NO.

REFS None

CALL	TP_DISC(conn)
ARGS	conn - in - connection number (Connno): specifies the connection number (CID) of the site for which a disconnect is requested.
FUNC	Requests a disconnect from a site. Checks the validity of the requested connection number (CID) and returns NO if not valid. Switches to shutting down STATE (TCPSHUT) if either input or output is active and returns YES. If no I/O is active, calls TCP_DISCONNECT to perform the actual disconnect and returns YES.
RETN	YES or NO.
REFS	TCP_DISCONNECT

CALL	TP_ID(site)
ARGS	site name - in - pointer to a character: specifies the site name for which the connection number is requested.
FUNC	Obtains the connection number (CID) of the requested site or 0 if there is no TCPCB for the site.
RETN	Connection number(CID) or 0.
REFS	TCP_FIND_SITE

2.2.1.5. Internal TCP Modules

2.2.1.5.1. Connects And Disconnects

These modules' names all begin with TCP_ and are only called within the TCP Routines. The routines are used to make the server connection and the client connections and disconnects.

CALL	TCP_CONNECT(p)
ARGS	p - in - pointer to a TCPCB.
FUNC	Creates a socket and attempts to connect to a remote site. Called from TP_CONN. Gets an I/O channel. Creates a socket and binds the channel number to the socket. Issues a connect request to the site whose address is specified in the TCPCB. This request is asynchronous and returns control to the program immediately. The module TCP_CLIENT is designated to receive the connection completion at a later time. Returns half connected locally (TCPHLOC) if successful, otherwise returns TCPFAIL and deallocates the TCPCB (TCP_PUT_ENT).
RETN	TCPHLOC or TCPFAIL.
REFS	TCP_PUT_ENT, TCP_CLIENT (asynchronous) and SYS$QIO

```
--------
CALL    TCP_CLIENT(p)   (asynchronous)

ARGS    p - in - pointer to a TCPCB.

FUNC    Receives control asynchronously when a connection request
        made by TCP_CONNECT completes.

        Checks the status and takes the following appropriate
        action.

        Status = SS$_NORMAL, then enters connected state
        (TCPCONN), informs TCPNIP of connection via the SERVICE
        module (ssap) and posts a read on the channel (TCP_READ).

        Status = ECONNREFUSED, then deassigns the channel, sets
        chan in TCPCB to 0 and leaves STATE as half connected
        locally (TCPHLOC).

        Status = [All Others], then deassigns the channel, prints
        error message, informs the TCPNIP of the connection
        failure (TCPFAIL) via the SERVICE module (ssap) and
        deallocates the TCPCB (TCP_PUT_ENT).

REFS    TCP_PUT_ENT, TCP_READ  and  SYS$DASSGN



--------
CALL    TCP_DISCONNECT(p)

ARGS    p - in - pointer to a TCPCB.

FUNC    Disconnects an active socket.

        Shutdown the socket (TCP_SHUTDOWN) unless in half
        connected locally mode (TCPHLOC) in which case there is
        no socket to shutdown.

        Deallocate the TCPCB (TCP_PUT_ENT).

RETN    Void

REFS    TCP_SHUTDOWN,  TCP_PUT_ENT
```

CALL	TCP_BIND()
ARGS	None

FUNC Creates a server socket and binds to the requested port.

Assigns a server channel.

Creates a socket using the IO$_SOCKET function.

Sets the socket option using the IO$_SETSOCKOPT function. The options set are:

    SO_KEEPALIVE
    SO_DONTLINGER
    SO_REUSEADDR

Binds the socket to the address specified in the TCPCB with the IO$_BIND function.

Specify the maximum queue length for a listen using the IO$_LISTEN function. Currently this is zero and can service one client at a time.

If any of the above I/O operations fail, the channel is deassigned, an error message is printed and the bad status returned.

If all goes well, normal status is returned.

RETN Status of the last I/O operation completed.

REFS SYS$ASSIGN, SYS$DASSGN, SYS$QIOW

```
--------
CALL    TCP_LISTEN()

ARGS    None

FUNC    Listens on the server channel for clients to request
        connection.

        Performs an asynchronous I/O on the server channel using
        the IO$_ACCEPT_WAIT function, setting TCP_WAIT as the
        module to be activated when a connection request is
        received.

        Checks status of the accept_wait and, if bad, prints an
        error message and deassigns the server channel.

        Returns status of I/O.

RETN    I/O status

REFS    TCP_WAIT  (asynchronous), SYS$DASSGN, SYS$QIO



--------
CALL    TCP_WAIT()   (asynchronous)

ARGS    NONE

FUNC    Called asynchronously when a connect request is seen for
        the server port.

        Checks the status of the I/O completion and if bad
        status, prints an error message, posts another listen
        (TCP_LISTEN) and returns.

        Assigns a new channel for the client socket.

        Performs an asynchronous accept of the connect request
        using the IO$_ACCEPT function, with which it specifies
        the server channel, the client channel, address of a
        location to receive the internet address of the remote
        station, and a module to be activated when the IO$_ACCEPT
        completes (TCP_ACCEPT).

        Checks the status of the I/O and if bad, prints a
        message, deassigns the client's channel and posts another
        listen (TCP_LISTEN).

REFS    TCP_LISTEN,   TCP_ACCEPT   (asynchronous)
        SYS$ASSIGN,   SYS$DASSGN,  SYS$QIO
```

```
--------
CALL    TCP_ACCEPT()   (asynchronous)

ARGS    None

FUNC    Processes a completed connection on the server channel.

        Checks the status of the IO$_ACCEPT and, if bad, prints
        an error message, deassigns the client's channel, and
        exits.

        Searches the TCP control tables for an entry with the
        internet address just received (TCP_FIND_ADDR) and
        activates a new TCPCB if not found (TCP_GET_ENT).

        If the STATE of this TCPCB is half connected locally
        (TCPHLOC), then the connection is now complete.  Change
        STATE to connected (TCPCONN), inform the TCPNIP via the
        SERVICE routine (ssap), record the client channel number
        and the internet address of the client in the TCPCB,
        initiate a read (TCP_READ) and a write (TCP_WRITE) on the
        client channel, and exit. The write is initiated in case
        the site had been fully connected earlier, had
        disconnected leaving queued output, and now has
        reconnected.

        If the STATE is shutting down (TCPSHUT), enter half
        connected remote (TCPHREM), post a read (TCP_READ) and
        exit.

        If the STATE is fully connected (TCPCONN) or half
        connected remote (TCPHREM), get a second TCPCB for this
        client and print a message warning of the second TCPCB
        for this client.

        If this is a new TCPCB, set STATE to be half connected
        remote (TCPHREM), record channel number, default SERVICE
        routine (as set in TP_START), internet address of the
        client, and sitename of the client (TCP_GET_SITE).  Post
        a read on the client's channel.

REFS    TCP_LISTEN,  TCP_FIND_ADDR,  TCP_GET_ENT,
        TCP_READ,    TCP_WRITE,      TCP_GET_SITE,
        SYS$DASSGN,  SYS$QIOW
```

2.2.1.7. Reads And Writes

All data communications to the TCP Transport layer are performed asynchronously. By agreement with the other TCPNIPs, all transmissions are preceded by the size of the data about to be transmitted. This size is transmitted as four bytes of binary data in network order, i.e. high order byte first. Since the VAX expects data in host order or low order byte first, all size data must be converted on the VAX.

On input, the size is read first followed by successive reads until all the data has been received. On output the size is transmitted first followed by successive writes until all the data has been transmitted. This requires three routines to accomplish each reception. The first routine posts a read of four bytes (TCP_READ). The second routine is activated when the reception of the four bytes completes. It posts a read to receive the indicated amount of data (TCP_READ_SIZE). The final routine is activated when the data reception is complete and calls itself asynchronously until all the data has been received (TCP_READ_DATA). A similar three routines are required for the transmission of size and data.

When an I/O error is encountered, it is assumed that the remote site has disconnected and appropriate action is taken (TCP_IO_FAIL).

The input buffers are obtained from a pool of buffers using CMBUFMGR. Output buffers are returned to this pool. Completed buffers are transferred to the calling program via the SERVICE Routine specified at connection time. Output buffers are obtained from the calling program and queued via the TP_OUT routine.

CALL	TCP_READ(p)
ARGS	p - in - pointer to a TCPCB.
FUNC	Posts a read for the first four bytes of incoming data. Checks the status of the TCPCB. Returns if the channel number is zero or input is already active. Posts a read to the channel using IO$_READVBLK for four bytes of size and specifying TCP_READ_SIZE as the module to be activated when complete. Checks the status of the read request and, if bad, prints an error message calls TCP_IO_FAIL to terminate the connection.
RETN	Void
REFS	TCP_IO_FAIL, TCP_READ_SIZE (asynchronous), SYS$QIO

CALL TCP_READ_SIZE(p) (asynchronous)

ARGS p - in - pointer to a TCPCB.

FUNC Checks the status of the size reception and starts the data reception.

Checks status of size reception and, if bad, writes an error message and calls TCP_IO_FAIL to terminate the connection.

Checks number of bytes received. If not the correct amount (four bytes), prints an error message and calls TCP_IO_FAIL to terminate the connection.

Converts size to host order.

Checks size of data coming. If size is greater than the largest buffer, prints a message and posts another read.

Gets an input buffer from the buffer pool (either CM_GETB or CM_GETL).

Sets the number of characters received to zero.

Posts a read using the IO$_READVBLK function, specifying the address of the text portion of the input buffer and the length of data expected, and specifying the TCP_READ_DATA routine to be activated when the reception completes.

Checks the status of the input request and, if bad, writes an error message and calls TCP_IO_FAIL to terminate the connection.

REFS TCP_IO_FAIL, TCP_READ_DATA (asynchronous), TCP_READ, SYS$QIO

CALL TCP_READ_DATA(p) (asynchronous)

ARGS p - in - pointer to a TCPCB.

FUNC Checks the status of the data reception and either transfers the message to TCPNIP or continues the reception as needed.

Checks status of data reception and, if bad, writes an error message and calls TCP_IO_FAIL to terminate the connection.

Adds the number of bytes received to the total previously received (set to zero in TCP_READ_SIZE).

Checks number of bytes received and takes the following appropriate action:

<u>Greater than Expected</u>: Prints warning message.

<u>Equal or Greater than Expected</u>: Subtracts length of the header (old transport header) from the total length. If shutting down STATE (TCPSHUT) and no output is pending, calls TCP_DISCONNECT to finish the disconnect and exits. Increments the incoming message counter (INCNT), records the connection number (CID) in the communication buffer, passes the communication buffer to TCPNIP via the SERVICE routine, sets the pointer to the communication buffer in the TCPCB (IB) to zero, posts another read (TCP_READ) and exits.

<u>Less than Expected</u>: Post a read using the IO$_READVBLK function, specifying the address of the next character position in the input buffer and the length data still expected and specifying the TCP_READ_DATA routine to be activated when the reception completes. Checks the status of the input request and, if bad, writes an error message and calls TCP_IO_FAIL to terminate the connection.

REFS TCP_IO_FAIL, TCP_READ_DATA (asynchronous), TCP_READ, TCP_DISCONNECT, SYS$QIO

```
CALL    TCP_WRITE(p)

ARGS    p - in - pointer to a TCPCB.

FUNC    Removes messages from the output queue and start
        transmission of the message size.

        Checks the status of the TCPCB.  Returns if the channel
        number is zero or output is already active.

        Removes an entry from the output queue and if null, exits
        after checking to shutting down mode (TCPSHUT).  If
        shutting down and input is inactive, calls TCP_DISCONNECT
        to finish the disconnect.

        Converts length of message to network order.

        Posts a write to the channel using IO$_WRITEVBLK of the
        four bytes of size and specifying TCP_WRITE_SIZE as the
        module to be activated when complete.

        Checks the status of the write request and, if bad,
        prints an error message calls TCP_IO_FAIL to terminate
        the connection.

RETN    Void

REFS    TCP_IO_FAIL,  TCP_WRITE_SIZE  (asynchronous)
        TCP_DISCONNECT,  SYS$QIO
```

CALL	TCP_WRITE_SIZE(p) (asynchronous)
ARGS	p - in - pointer to a TCPCB.
FUNC	Checks the status of the size transmission and starts the data transmission. If the channel is zero or output is inactive, exits. Checks status of size transmission and if bad writes an error message and calls TCP_IO_FAIL to terminate the connection. Checks number of bytes transmitted. If not the correct amount (four bytes), prints an error message and calls TCP_IO_FAIL to terminate the connection. Converts size back to host order. Sets the number of characters transmitted to zero. Posts a write using the IO$_WRITEVBLK function, specifying the address of the text portion of the output buffer and the length data to transmit, and specifying the TCP_WRITE_DATA routine to be activated when the transmission completes. Checks the status of the output request and, if bad, writes an error message and calls TCP_IO_FAIL to terminate the connection.
REFS	TCP_IO_FAIL, TCP_WRITE_DATA (asynchronous), TCP_WRITE, CM_GETB, CM_GET, SYS$QIO

CALL TCP_WRITE_DATA(p) (asynchronous)

ARGS p - in - pointer to a TCPCB.

FUNC Checks the status of the data transmission and either returns the buffer to the buffer pool or continues the transmission as needed.

Checks status of data transmission and, if bad, writes an error message and calls TCP_IO_FAIL to terminate the connection.

Adds the number of bytes actually transmitted to the total previously transmitted (set to zero in TCP_WRITE_SIZE).

Checks number of bytes transmitted and takes the following appropriate action:

Greater Than Expected: Prints warning message.

Equal Or Greater than Expected: Increments the outgoing message counter (OUTCNT), informs TCPNIP of the output completion via the SERVICE routine, returns the output buffer to the buffer pool (CM_PUTB), sets the pointer to the communication buffer in the TCPCB (OB) to zero, posts the next write (TCP_WRITE) and exits.

Less Than Expected: Posts a write using the IO$_WRITEVBLK function, specifying the address of the next character position in the output buffer and the length data still to be transmitted, and specifying the TCP_WRITE_DATA routine to be activated when the transmission completes. Checks the status of the output request and if bad writes an error message and calls TCP_IO_FAIL to terminate the connection.

REFS TCP_IO_FAIL, TCP_WRITE_DATA (asynchronous), TCP_WRITE, CM_PUTB, SYS$QIO

2.2.1.5.3. SERVICE

CALL TCP_IO_FAIL(p)

ARGS p - in - pointer to a TCPCB.

FUNC Handles the condition when the I/O has failed and we assume that the remote TCPNIP has disconnected.

If output is active, re-queue the output buffer for later transmission if the remote TCPNIP reconnects.

If input is active, return input buffer to buffer pool (CM_PUTB).

If fully connected (TCPCONN), shutdown the connection (TCP_SHUTDOWN) and change to half connected locally (TCPHLOC) STATE. Inform the TCPNIP of the remote disconnect via the SERVICE routine (ssap), print a warning message, and exit.

If shutting down (TCPSHUT) or half connected remote (TCPHREM), disconnect completely (TCP_DISCONNECT).

RETN Void

REFS TCP_SHUTDOWN, TCP_DISCONNECT, CM_PUTB, SYS$QIOW

CALL TCP_SHUTDOWN(p)

ARGS p - in - pointer to a TCPCB.

FUNC Shuts down a socket.

Issues an I/O request using the IO$_SHUTDOWN function forbidding any future sends or receives.

Checks the status of the I/O request and, if bad, prints a warning message.

Deassigns the client's channel.

Sets the channel recorded in the TCPCB to zero and returns.

RETN Void

REFS SYS$QIOW, SYS$DASSGN

CALL	TCP_GET_ENT(p)
ARGS	p - out - pointer to a pointer to a TCPCB.
FUNC	Gets an available TCPCB from the TCP control table. Searches the TCP control table for an TCPCB with a connection number (CID) of zero. Sets the connection number (CID) in the first available TCBCB to the index of the TCBCB in the control table + 1. Sets the argument p to the address of the TCPCB and returns YES. Prints an error message and returns NO if no TCPCBs are available.
RETN	YES or NO
REFS	None

CALL	TCP_PUT_ENT(p)
ARGS	p - in - pointer to a TCPCB.
FUNC	Deallocates a TCPCB in the TCP control table. If input and/or output is active, returns the communication buffer to the buffer pool (CM_PUTB). Empties the output queue and returns all buffers to the buffer pool (CM_PUTB). Deassigns the I/O channel. Sets the connection number (CID), input count, output count, state, channel, remote address, pointer to input and output buffers and the SERVICE routine (ssap) to zero. Sets the site name to all blanks.
RETN	None
REFS	SYS$DASSGN

CALL TCP_FIND_ADDR(addr,p)

ARGS <u>addr</u> - in - struct sockaddr_in and specifies the internet address of remote TCPNIP.

<u>p</u> - out - pointer to a pointer to a TCPCB.

FUNC Searches for a TCBCB with the requested internet address.

Searches through the TCP control table for the requested address (addr argument).

If found, sets pointer p to the address of the TCBCB and returns YES.

If not found, returns NO.

RETN YES or NO

REFS None

CALL TCP_FIND_SITE(site,p)

ARGS <u>site</u> - in - pointer to a character. Specifies the site name of the remote TCPNIP.

<u>p</u> - out - pointer to a pointer to a TCPCB.

FUNC Searches for a TCBCB for the requested TCPNIP.

Searches through the TCP control table for the requested sitename (site argument).

If found, sets pointer p to the address of the TCBCB and returns YES.

If not found, returns NO.

RETN YES or NO

REFS None

CALL TCP_GET_ADDR(site,addr)

ARGS site - in - pointer to a character. Specifies the site name of the remote TCPNIP.

addr - out - pointer to integer which specifies the internet address of remote TCPNIP.

FUNC Finds the internet address of the requested site.

Searches the table SITENAME which was set up by NETGEN in TP_START, for the requested sitename.

If found, sets the argument addr equal to the address of the internet address found in the table ADDRESS, also set up by NETGEN, at the same index as the requested sitename. Returns YES.

If not found, prints an error message and returns NO.

RETN YES or NO

REFS None

CALL TCP_GET_SITE(addr,site)

ARGS addr - in - integer and specifies the internet address of remote TCPNIP.

site - out - pointer to a character. Specifies the site name of the remote TCPNIP.

FUNC Finds the sitename for the requested internet address.

Searches the table ADDRESS which was set up by NETGEN in TP_START, for the requested internet address.

If found, sets the argument site equal to the address of the sitename found in the table SITENAME, also set up by NETGEN, at the same index as the requested address. Returns YES.

If not found, prints an error message and returns NO.

RETN YES or NO

REFS None

2.2.2. Sun Microsystems Implementation

The Sun version of integrating TCP/IP network software into the AMRF Network Interface Process (NIP) was created by replacing the existing NIP software from the transport layer down (transport, network, link and physical layers) with calls to the UNIX 4.2 BSD system-provided TCP network services. The layers above transport were not affected (copymail).

2.2.2.1. Comparison With The Sun NIP Serial Version

As noted above, the TCP/IP protocol used over Ethernet logically corresponds to the serial NIP's physical through transport layers. In developing the TCP/IP version, efforts were made to keep all common modules between the two versions identical. This was not always possible, and some minor differences exist. In summary:

Modules in serial NIP that are not in TCP NIP:
- io.c
- loader.c
- mlink.c
- net.c
- netgen.c
- slink.c
- xport.c

Modules common to both NIPs that are identical:
- cmbufmgr.c
- timer.c
- dispatch.c
- sfuncs.c

Modules common to both NIPs that are NOT identical:

copymail.c - The differences here are trivial and the two could (and should) be made identical. However there are two differences. The following line has been commented out of the getcmd() routine in the serial version:

```
if (cmd.cmd_len == 0) cmd.cmd_act = NIPNOP;
```

The TCP version prints out a message to the standard error if it receives a mailgram for which it has no entry in the mail delivery table. This is quite helpful in flagging bugs that would otherwise go undetected.

nipmain.c -

1) The TCP version is primed to receive it's NIPCMD commands from the station "TVAX" instead of "VAX". This was done since there will be two separate NIPs running on the AMRF VAX.

2) The default TCP NIP server port number (1526) can be changed by entering the -t option on the command line.

3) commented out of the tcp version are:
   - ability to set def_nadd[] from the command line ... it doesn't apply here
   - calls to io_ini(), net_ini()
   - calls to initialize the loader.c routines ... they weren't used here.

2.2.2.2. Possible Integration With Sun Serial NIP To Create A Single Sun NIP

This version could be combined with the serial link version of the Sun NIP by inserting a "transport interface" layer under copymail and having it (the transport interface layer) determine which transport, AMRF transport or TCP, should be used. Other things that would have to be done in order to effect this transport interface layer:

1) Transport interface layer will keep track of connection numbers and map to/from a connection number that is meaningful to the particular transport layer. The transport interface layer will also keep track of connection states in the simplest context. This is to keep track of live connections, clear its table when a disconnect occurs, when errors occur, etc.

   Alternatively, one could put a "transport type" field in the mail delivery table entry in copymail and reference that in order to determine which transport layer to call. One must be careful in this case to not confuse the cid of one transport layer with the cid of the other transport layer. Check the use of cid throughout copymail and check for the necessity to also use check on "transport type" to keep mailboxes pointing to the right transport layer. This approach modifies copymail but does not create an intermediate "transport layer interface" layer. Unfortunately, this also requires changing the MDT entry structure, and mdtent is an integral part of the mailbox manager (mgrcmd) data structure from netcmd. Therefore when copying from the mgrcmd copy to the mdtent copy, we will have to do it one field at a time when making new MDTENTS in the table (MDTCONN).

2) A combined version of SIGIO signal handler interrupt handler will have to be created. It will decide (based upon the file descriptor) whether the "serial link" handler should be called or whether the "TCP" handler should be called or BOTH!

   Have TCP and xport(or io.c) export a Boolean that is TRUE if that module has file descriptors open and only call the sigio handler if the variable is TRUE for that module.

```
    i.e. sigio_handler() {
                 if (serial_link_fds_active) serial_sigh();
                 if (tcp_fds_active) tcp_sigh();
         }
```

2.2.2.3. TCP Stream Sockets Used

TCP stream sockets provided by Berkeley Unix are used as the transport mechanism. These provide a reliable byte stream path of communication between the two transport endpoints. In addition, the stream library (/usr/lib/libstream.a) provided initport() is used for opening and initializing the sockets. The sized_io.c routines of sized_read() and sized_write() are not used directly from the library for reasons that are pointed out later.

2.2.2.4. Algorithm for Establishing a TCP Connection

2.2.2.4.1. Socket Connections And Client/Server Relationships

TCP sockets, once established, are bi-directional, full-duplex, and balanced. However, the connection setup procedures are not balanced but follow a client/server model. In this model, the server provides a certain service to client processes. Therefore for purposes of establishing the TCP connection, one process must be the server and the other process must be the client.

In the client/server model, a server process typically passively listens for connection requests at a "well-known" port address. The client actively connects to the "well-known" server address to establish the connection. The problem with this model is that is doesn't fit the AMRF NIP architecture. Two NIPs communicating in the AMRF are equal entities and communicate in a symmetric manner.

We overcome this problem by assuming the client role in connection establishment. If the active connect attempt fails, we switch and become the server waiting for a connection request from the remote system. Upon startup, the NIP begins listening for connection requests from other NIPs at a "well-known" NIP server socket number (NIP_PORT). If a connection request arrives at the socket from a known host it is accepted and entered into the transport connection table. When instructed by session layer to make a new connection, the NIP will first check to see if the desired connection is already in the transport table. If so, that socket is used. If not, transport will attempt to create a connection to the host by connecting as a client to the remote NIP's server port. If successful .... great. If unsuccessful due to the host not being up (ETIMEDOUT) or the server not active (i.e. it's NIP isn't running -- ECONNREFUSED), transport makes the table entry anyway assuming the remote host will eventually come up and do an active connect to the local host upon direction of it's session layer. In this case, we pass DONE back to session and for all it knows, the connection is complete. If the local connection attempt is unsuccessful due to other problems, the error is passed back to session and the entry is not made in the table.

This method can be used due to the nature of how session layer connections (mailboxes) are made. Both hosts are explicitly told via their NIPCMD mailboxes to make each mailbox and therefore, each transport connection. If the session layer changes it's connection procedures, the TCP connection algorithm may have to be changed also.

2.2.2.4.2. TCP Connect Timeout

When a client attempts to connect to a station where the NIP server is not running, connect() will return immediately with errno == ECONNREFUSED. But if a client attempts to connect to a station that is down, the connect must timeout before it returns with errno == ETIMEDOUT. The timeout period for connection attempts in TCP is 45 seconds. Sun Microsystem does not provide the user with the flexibility to change to default timeout value. One cannot get hold of the system's socket structure to make the change. Sun claims that a nonstandard timeout value for TCP make it "not TCP".

Attempts were made to get around this by checking to see if the remote station was up by use of (among other things) the up() routine in the ping.c module. This code was patterned after the ping(8) utility in the UNIX manual. No attempt proved totally successful. Therefore the 45 second timeout remains for the time being ... a minor annoyance.

2.2.2.5. TCP Socket Disconnection

Unfortunately, the Sun TCP does not provide unambiguous notification when the remote system closes (disconnects) the socket. This is confusing because it does not provide the programmer with the ability to distinguish between the cases when

- a) the remote host actively disconnects the socket and
- b) the remote host process died in some very bad manner, such as a system crash.

When the remote site disconnects or dies, the local NIP notices it by either

1) a -1 return on read,
2) a 0 return on read or
3) a broken pipe error on writing.

Since there is no way to distinguish bad from good closes of a transport socket, all of these cases are treated as active (good) disconnections by the remote NIP.

2.2.2.6. Configuration Table

The configuration table that is used to map the AMRF site name to the TCP recognized host name is kept in tcp.c along with the protocol code. Logically this is a replacement for the old config.c configuration tables in the serial version of the NIP. It has been proposed that the use of the "yellow pages" service on the Suns could provide this mapping via the host alias capability. This would eliminate the need for the configuration table host_tab[] and the mapping routines host_to_site() and site_to_host(). The AMRF network configuration responsibilities would then fall partially on the Sun system administrator to maintain the yellow pages directory.

2.2.2.7. NIP Status Packet Remains Unchanged

The NIP status structure was kept the same as in the old serial NIP to maintain compatibility with every other NIP in the AMRF and with what the network manager process NETCMD expects to receive. In particular, some of the fields of the link status (cmlkst) structure have no application in the TCP NIP but are retained and filled with zeroes.

2.2.2.8. TCP CONNECTION STATES

Following are the states of the TCP connections. The state is really kept as a composite of two state variables:

(1) the actual TCP connection to the remote host (R) and
(2) the state of the connection as viewed by the local session layer software (L).

Both variables may be in either the connected state (C) or the disconnected state (D). In addition, the remote state variable may be in the shutdown (S) state when flushing output data prior to disconnecting. Therefore there are five composite states: D/D, D/C, C/D, C/C and D/S expressed in <Local>/<Remote> format.

Note that D/D signifies that both local and remote variables are in the disconnected state which is equivalent to a nonexistant connection (i.e., there is no entry in the connection table if D/D is the IN state and the entry is deleted if D/D is the OUT state). The "?" state is a don't care in the table below.

Table V-4. Sun TCP NIP: Change Of State As A Function Of Events.

Event	IN STATES L/R	Action/Comments	OUT STATES L/R
User conn	D/D	Attempt to connect to remote host is successful	C/C
User conn	D/D	Attempt to connect to remote host is unsuccessful	C/D
User conn	C/D	Return "DONE". No action taken.	unchged
User conn	C/C	Return "DONE". No action taken.	unchged
User conn	D/C	Connection is now complete.	C/C
User conn	D/S	Keep the connection after all.	C/C
Remote conn	D/D	Accept connection	D/C
Remote conn	C/D	Accept connection	C/C
Remote conn	?/C	ERROR. Ignore or refuse.	unchged
Remote conn	D/S	ERROR. Ignore or refuse.	unchged
User disconn	C/C	No output is queued. Disconnect from remote.	D/D
User disconn	C/C	Output data is queued. Enter "SHUTDOWN" state until output is all transmitted.	D/S
User disconn	D/?	ERROR.	unchged
Remote Disc	C/C	Keep table entry. No notification sent to session.	C/D

Table V-4 -- Concluded

Remote Disc	D/C	Delete entry.	D/D
Remote Disc	D/S	Send OUTCAN to session for the data buffers not transmitted.	D/D
Remote Disc	?/D	impossible event.	---
User data	C/C	Send the data to the remote	unchged
User data	C/D	Queue the data to the remote	unchged
User data	D/?	Attempt to send an unconnected transport. Send OUTCAN to session.	unchged
Remote data	C/C	Send the data to the session	unchged
Remote data	D/C	data is queued in the pipe by not reading it yet	unchged
Remote data	D/S	ignore incoming data from remote	unchged
Remote data	?/D	impossible event.	---
last data xmitted from output queue	D/S	output finished draining	D/D

VI. Operator Reference Section

This section identifies and describes the steps that must be followed in order to start up and shut down the AMRF network. Although these steps are accurate for the network topology existent during 1986, it is expected that changes in shop floor equipment and upgrades in network interfaces will result in a modified operators guide. Do not assume that the instructions in the following sections are applicable to the "current" network topology.

1. THEORY OF OPERATIONS

1.1. Selecting The Network Configuration

The specific steps that must be performed to activate the network are determined by the network configuration that is to be run. Figure VI-1 shows the current network topology. Table VI-2 identifies the connections between workstations and the network services necessary to support them.

In the current network topology (Section II.3.1), there are two basic subnetwork environments: the asynchronous RS232 subnetwork, and the Ethernet TCP/IP subnetwork. The Ethernet TCP/IP subnetwork environment is further divided into the VAX-based environment and the front end common memory-based environment (labeled "SUN-FE", below). If the anticipated node-to-node interactions (i.e., the configuration to be tested) are solely within a single environment, then there is no need to activate nodes in any of the other environments. Table VI-1 lists the workstations connected through these environments.

Network connections between any two of these environments assume the existence of common memory in each of these environments. In virtually all implementations, except one, the common memory area automatically exists before the network attempts to access it. It is either created when the user process starts (VAX and HP-based implementations), or it exists in hardware (multibus-based implementations). The exception is the Sun Microsystems-based implementations, where the common memory is implemented as a separate process (i.e., program) that must be run before the NIP is started.

Table VI-1. Workstations - Listed By Subnetwork Connection

SUBNETWORK	SUBSET	WORKSTATIONS
RS232	-	ATC, CMM, HGP, HMB, HMC, HRC, HVS, HWS, IRC, IWS, SRI, TWS, VAX
ETHERNET	SUN-FE	CDWS, CELL, MHS, PPL, VWS
	VAX	VAX

Table VI-2. Table Of Network Linkages And Required Network Services

	ATC	CDWS	CELL	CMM	HGP	HMB	HMC	HRC	HVS	HWS	IRC	IWS	MHS	PPL	SRI	TWS	VAX	VWS
ATC	---	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
CDWS		---	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
CELL	B	B	---	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
CMM				---	...	...	...	...	...	...	...	...	...	...	...	...	...	...
HGP					---	...	...	...	...	...	...	...	...	...	...	...	...	...
HMB						---	...	...	...	...	...	...	...	...	...	...	...	...
HMC							---	...	...	...	...	...	...	...	...	...	...	...
HRC					D/F			---	...	...	...	...	...	...	...	...	...	...
HVS								I	---	...	...	...	...	...	...	...	...	...
HWS			BGD		D/F	D/F	D/F	D/F		---	...	...	...	...	...	...	...	...
IRC											---	...	...	...	...	...	...	...
IWS			BGD	DEH							DEH	---	...	...	...	...	...	...
MHS			BC			DGC			DGC				---	...	...	...	...	...
PPL														---	...	...	...	...
SRI												DEH			---	...	...	...
TWS	D		BGD													---	...	...
VAX	E	AG	FB	D	D	D	D	D	D	D	D	D	E	FA	D	E	---	...
VWS			B														FA	---

Key to Services:

A - Common memory active on front end processor
B - 'A' + communication process to CELL PC
C - 'A' + communication process to MHS PC
D - Serial NIP connection(s) to/thru VAX
E - Serial NIP connections between nodes
F - Ethernet NIP connections between nodes
G - Ethernet NIP connection to VAX
H - Serial NIP Connections go through IWS
I - Direct, non-network connection

1.2. Determining The Required Network Services

Once you have determined the stations that you wish to interconnect in the network configuration, you must now determine how the interconnection will occur. Table VI-2 identifies the network services that are required in order to establish a link between any two stations in the network. (The commands necessary to connect the mailboxes are not identified, but will be discussed in Section VI.1.3.)

Some examples of determining the required network services are listed below:

EXAMPLE 1: determine what is needed to support a link between the Cleaning and Deburring Workstation (CDWS) and the CELL. Figure VI-1 indicates that the CELL and the CDWS are both attached to the same front end Sun Microsystems common memory server. Table VI-2 indicates that the required configuration supporting the links between these two systems consists of: creating the common memory with which and thru which both systems exchange data, and running the CELL communications process that interfaces the CELL to the common memory.

EXAMPLE 2: determine what is needed to support a link between the Inspection Workstation Controller (IWS) and the IMDAS (on the VAX). Figure VI-1 indicates that IWS and the VAX are remotely located, so network services (NIPs running on each computer system) are required to create the point-to-point connection between the VAX and the IWS. Per the previous discussion about common memory activation (Section VI.1.1.), nothing needs to be done to specifically activate either of these common memories. Table VI-2 agrees with this diagnosis: it indicates that NIPs have to be active on both the IWS and the VAX.

EXAMPLE 3: determine what is needed to support a link between the IWS and the CELL. Figure VI-1 indicates that the data path would extend from the IWS through the VAX (using asynchronous RS232 connections) and over the ethernet links to the common memory front end. This immediately indicates that network services are necessary between IWS and the VAX, and between the VAX and the front end common memory server to which CELL is attached. Table VI-2 indicates that common memory must be active on the CELL's front end system, the CELL communications program must be active in order to link the CELL to its common memory, the Ethernet NIPS must be active on the front end machine and the VAX, and the serial NIPS must be active on the IWS and the VAX in order to provide end-to-end connectivity.

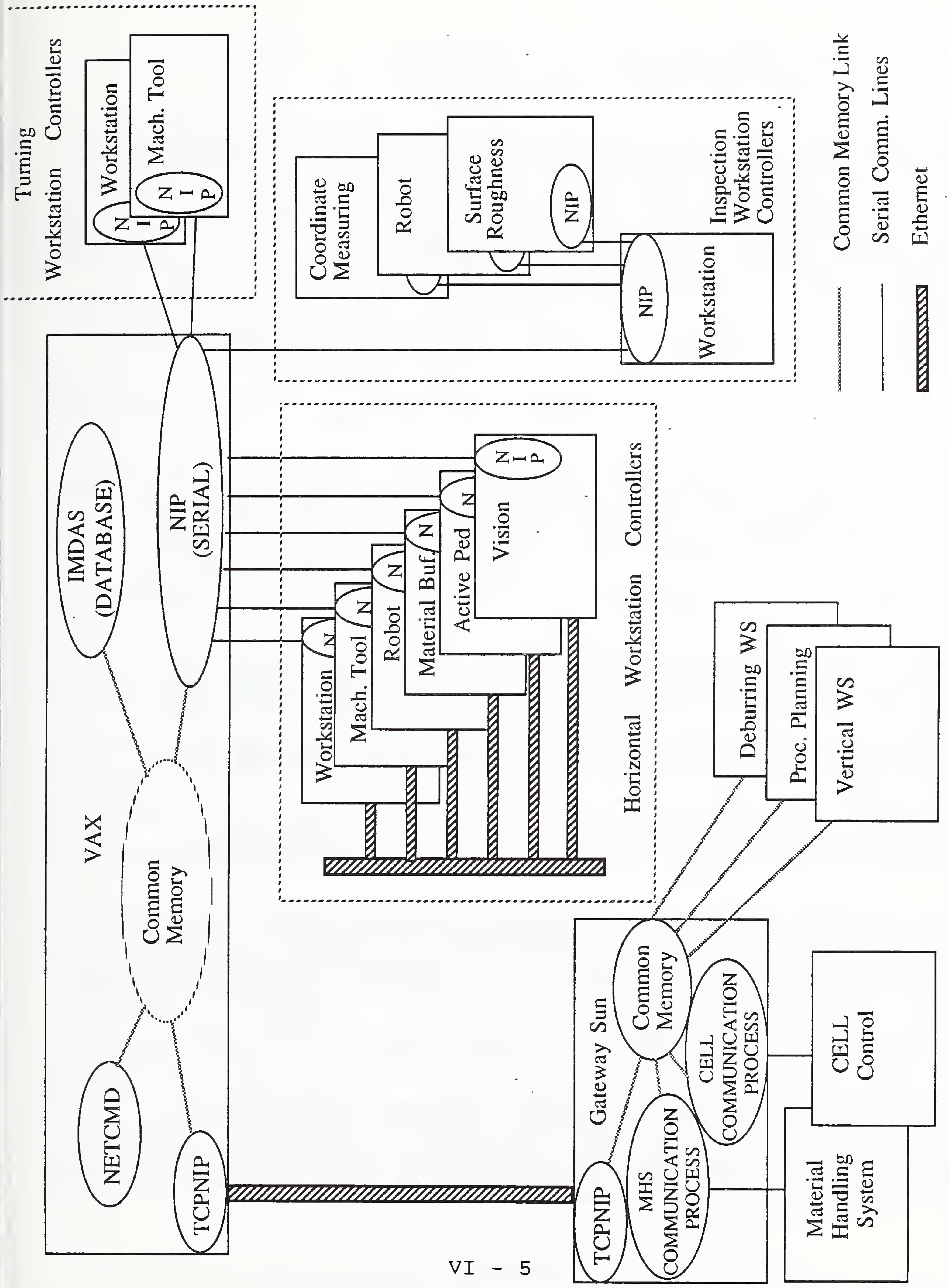

Figure VI-1. The Topology of the 1986 AMRF Network

1.3. Script Files

1.3.1. Purpose of Script Files

Once the NIPs have been started and the network manager program (NETCMD) is running, it is desirable to connect specific common memory mailboxes between one or more systems. This is done by issuing CONNECT commands to NETCMD. These commands have a specific structure that is not especially easy to type. Although the network connections can be dynamically made (and broken), we tend to make the same connections over and over. In order to expedite the network connection process and reduce the potential for operator error, we use an editor to create a series of CONNECT (or DISCONNECT) commands in a file, and then reference the file name when building the network connections.

1.3.2. Script File Naming Conventions

The file names follow a general convention that uses a two-part name. The first part of the name is descriptive of the function of the file contents and also identifies the workstation or service to which it pertains. The second part of the name, termed the "file name extension", is always set to "NET" to distinguish it as a network connection file. The first part of the file name is separated from the second part with a period. EXAMPLE: IWSbase.net

[NOTE: File names on the VAX, where the primary network manager program (NETCMD) is installed, are case insensitive. That is, it does not matter if they are stored or accessed in UPPER or lowercase on the VAX. However, NETCMD on the Sun Microsystems computer IS case sensitive: all names must be entered in lowercase.]

The network connections fall into two major categories: connections to NETCMD, and connections to other stations or services (e.g., IMDAS).

Each station that has an operating NIP must connect to NETCMD before it attempts to make connections to other stations or services across the network. These initial connections are basic to the operation of the station and serve as a conduit through which the additional connections are made. Script files that contain these basic CONNECT commands have the word "base" as part of their file name. EXAMPLES: IWSbase.net, TWSbase.net, etc.

Script files that contain CONNECT commands for mailbox connections between two workstations (or a workstation and a remote service) identify both ends of the connection in the file name, and place the identification of the initiator of the inter-station dialogue first. Some examples are:

HRCHGP.net	- indicates this file contains one or more commands connecting HRC to HGP. HRC initiates the dialogue between the two.
CELLIWS.net	- indicates this file contains one or more commands connecting CELL to IWS. CELL initiates the dialogue between the two.
IWSDB.net	- indicates this file contains one or more commands connecting IWS to the IMDAS (on the VAX). IWS initiates the dialogue between the two.

2. STARTING THE NETWORK

The following subsections detail the startup procedure used to bring up all or a portion of the AMRF workstations and services. The procedure is presented in two formats:

(1) A discretionary procedure. This format provides a more detailed description of the specific steps in the procedure, and identifies points where the operator must make a decision that depends on the specific network configuration being booted.

(2) A streamlined procedure. It optimizes the sequence of the steps to bring up the network configuration as rapidly as possible. It assumes that the entire network is to be booted, and that you are fully familiar with the display screens and terms used in the boot process.

Other network startup procedures have been used during the evolution of the AMRF. Their use is determined by the availability of other computers, other circuit routing. They are only used if the VAX is unavailable. Using the VAX provides the easiest, most cohesive network startup, so only the VAX procedure is described below.

These sections assume that the operator is reasonably familiar with the Digital Equipment Corporation VAX/VMS operating system command language (DCL), with UNIX, and the Sun Microsystems computers (hereafter referred to as 'Suns'), including their windowing environment.

The terms 'DEMO', 'VAX' and 'window' are used throughout the procedure. 'VAX' refers to the AMRF VAX 11/785 operating under the VMS operating system. DEMO refers to the Sun Microsystems computer functioning as the front end common memory server identified in Figure VI-1, operating under the (4.2 BSD) UNIX operating system. 'Window' refers to a delineated portion of the Sun video display screen that functions as a "terminal screen" for the application active within that delineated portion of the screen, much the same as the standard terminal screen.

The login procedure and the mailbox connection procedure assume that the user will ALWAYS use the DEMO Sun. That is, DEMO will be used to connect the clientele for which it serves as a common memory front end, or it will be used (additionally or in place of) as a terminal interface to the VAX computer.

2.1. The Discretionary Startup Procedure

Use this procedure if you only want to boot a subsection of the AMRF network, or if you want additional details of the network startup process.

An attempt has been made to make this procedure sufficiently descriptive so that decision points (where you have options that depend on the network configuration that is being booted) are clearly identified.

2.1.1. Login

(1) Login to DEMO from the console keyboard with user name AMRFTEST (no password is required). The login procedure checks if you are operating from the DEMO console keyboard, and, if so, issues command "SUNTOOLS VAXNET". This sets up the windows for :

CELL	- for the communications process that links the PC-resident CELL and the local common memory.
MHS	- for the communications process that links the PC-resident MHS and the local common memory.
NETCMD	- for login to the VAX and display of a very wide window into which all network status displays will fit (192 characters wide).
CMM	- for the local common memory process.
TCPNIP	- for the local network interface process that operates over the ethernet TCP/IP link to the VAX.
SCRATCH	- for optional, discretionary use.

It also starts the local common memory manager (CMM) and the TCPNIP. The wide NETCMD window has been programmed to perform a TELNET to the VAX, so use the mouse to place the arrow into that window and get ready to login to the VAX when the "Username:" prompt is displayed. Login with user name AMRFINT1 (password is available on a "need to know basis" from the AMRF configuration manager.)

Login to the VAX in the NETCMD window only if you are about to configure a network that requires the use of the VAX to initiate mailbox connections (and to act as a router link between remote systems). If you do not login to the NETCMD window before the VAX times out the user login, the window will close (disappear). If you need to bring back the window, use the mouse to display the Suntools menu, select the network manager submenu, and select the "NETCMD" option. The window will be recreated and the automatic VAX "Username:" prompt will appear as before.

(2) If you are starting the entire AMRF network, or if you are configuring network connections that utilize the VAX, login, on a nearby VT100 terminal to the VAX, as AMRFINT1 and follow the prompt to the [AMRFINT1.NET] subdirectory.

2.1.2. Establish The Environment

(1) If you are configuring network connections in the 'A' category, as identified in Table VI-2, then you are finished.

(2) If you are configuring network connections that require the addition of category 'B' (CELL communications link), as identified in Table VI-2, then, in the CELL window, issue commands

(a) if the debugger is NOT required (i.e., you are not looking for program errors):

```
su - wenger
cd cmcell/cell
cell
<an extra carriage return>
```

(b) if the debugger is required:

```
su - wenger
cd cmcell/cell
dbx sun_tty01
run 17 0
<an extra carriage return>
```

(3) If you are configuring network connections that require the addition of category 'C' (MHS communications link), as identified in Table VI-2, then, in the MHS window, issue commands

(a) if the debugger is NOT required:

```
su - wenger
cd cmcell/mhs
mhs
<an extra carriage return>
```

(b) if the debugger is required:

```
su - wenger
cd cmcell/mhs
dbx sun_tty02
run 17 0
<an extra carriage return>
```

2.1.3. Start The NIPs

The network interface processes (NIPS) for all of the computers in the Inspection Workstation cluster (IWS, CMM, SRI, IRC) and the front end common memory computer, called DEMO, (for VWS, PPL, CDWS, CELL and MHS) will automatically be started when the station software is executed, and therefore require only some coordination between the network operator and the workstation operator to indicate that the workstation has been initialized before the network boot process can continue.

The remaining NIPs require some manipulation by the network operator in order to make them operational and able to process network packets. This manipulation is detailed in Section VI.2.4.1. of this document. You may proceed to initialize those NIPs, assuming they are required for your network configuration, without waiting for the IWS initialization process to complete. However, a general rule is: a workstation NIP must successfully complete initialization before a command is issued to NETCMD requesting a mailbox connection through that NIP.

2.1.4. Connect Common Memory Mailboxes

In DEMO's NETCMD window, issue the following commands (without a leading '@' sign):

NETNAMES	(establishes the network logical names)
NETSTART	(starts MBHAND and the SERIAL NIP in the background; log file created)
TCPNIP	(used only if DEMO will be connected, runs the NIP in the foreground; displays status of all TCP NIP transactions; requires dedicated use of the terminal)
NETCMD 200	(brings up the network manager display in anticipation of mailbox connections)

2.1.4.1. Complete Set Of Mailbox Connections

Script files (see Section VI.2.1.3) have been created to simplify and expedite the mailbox connection and network startup process. The following list identifies the names, syntax and sequence in which the script files must be submitted to NETCMD. Subsequent script files assume that the previous script files have been processed normally.

Enter these script files (with the "@" sign prefixed) at the NETCMD "Command:" prompt.

NOTE: ALWAYS issue the xxxBase.net script file BEFORE any other script files for that station. If two stations are to be interconnected, BOTH of their xxxBase.net script files must be processed before any of their other mailboxes can be connected.

```
                         @vwsbase
                         @hwsvall
                         @twsvall
                         @hvsbase


                         @iwsbase
                         @iwssri
                         @iwscmm
                         @iwsirc
                            ....  or use @IWSALL.NET to get all
                                  these IWS script files at one
                                  time


                         @hwsdb    (NOTE: these files are only
                         @hmcdb     used if the IMDAS is to
                         @hrcdb     accessed.)
                         @twsdb
                         @ppldb
                         @atcdb
                         @dwsdb
                         @vwsdb
                         @iwsdb
                         @hvsdb
                            ....  or use @ALLDB.NET to get all
                                  these IMDAS script files at
                                  one time


                         @cellhws    (NOTE: these files are only
                         @celltws     used if the Cell is to be
                         @celldws     included in the active
                         @cellvws     configuration.)
                         @celliws
                            ....  or use @CELLALL.NET to get all
                                  these CELL script files at one
                                  time

                         @mhshvs
                         @mhshmb
```

2.1.4.2. Discretionary Mailbox Connections

Appendix E lists all (currently) available script files. Section VI.1.3.2 describes the script file naming convention. With that information, an operator can start any network configuration.

Just a reminder: ALWAYS issue the xxxBase.net script file BEFORE any other script files for that station. If two stations are to be interconnected, BOTH of their xxxBase.net script files must be processed before any of their other mailboxes can be connected.

2.2. The Streamlined Startup Procedure

Use this procedure only if you intend to boot the entire network configuration or if you are looking for shortcuts to expedite your network boot process.

If you desire more detailed information for the boot procedure, read Section VI.2.1.

This procedure optimizes the sequence of the steps to bring up the network configuration in order to accomplish it as rapidly as possible. It assumes that the entire network is to be booted, and that you are fully familiar with the display screens and terms used in the boot process.

Perform the following steps in sequence. Subsequent steps (with the exception of steps 1 and 2) assume the completion of all preceding steps.

(1) Login to DEMO from the console keyboard with user name AMRFTEST (no password is required). The login procedure issues command "SUNTOOLS VAXNET" for you and sets up the windows for CELL, MHS, NETCMD, CMM, TCPNIP, and a spare window for optional use.

The local common memory manager (CMM) and the TCPNIP will start automatically. The wide NETCMD window will TELNET to the VAX, so use the mouse to place the arrow into that window and get ready to login to the VAX when the "Username:" prompt is displayed. Login with user name AMRFINT1 (password is available on a "need to know basis" from the AMRF configuration manager.)

(2) While you are waiting for step (1) to get to the "Username:" prompt in the NETCMD window, use a nearby VT100 terminal and login to the VAX as AMRFINT1. At the first prompt, enter "NET". This will place you into the [AMRFINT1.NET] subdirectory.

If the "Username:" prompt appears in the NETCMD window before you finish your login on the VT100, be sure and stop what you are doing on the VT100 and login in the NETCMD window and then resume on the VT100. If the NETCMD login times out (you have approximately 30 seconds), then NETCMD window will close and you will have to regenerate it using the Suntools menu (select the NETMGR submenu and then the NETCMD option).

(3) On the VT100 terminal, issue the following commands (without a leading '@' sign):

```
NETNAMES
NETSTART
TCPNIP
```

(4) Press the RESET buttons for all HWS and TWS controllers. (i.e., HWS, HMB, HMC, HRC, HGP, HVS, TWS, and ATC)

(5) On DEMO, in the NETCMD window, issue the following commands (without a leading '@' sign):

```
ALLDOWN       (performs sequential @DOWNs)
```

"ALLDOWN" performs an @DOWN on all the stations of the AMRF. You will see the name displayed for the station whose NIP you are being asked to start. Press RETURN or ENTER before entering the boot commands in order to see if the controller interface is active. You should see a ">" in response to pressing RETURN.

For HWS, HMC, HRC, HGP, and HMB the sequence is (uppercase only):

```
CAL FE2000
GO
```

For HVS the sequence is (uppercase only):

```
CAL FE2000
SR 2700
PC 1000
GO
```

For TWS and ATC, the sequence is:

```
g980000
r2
2700
r3
1000
go
```

If one (or more) of the controller NIPs does not respond to your RETURN, then verify that the circuit is operational and fix it, if inactive (Section VI.2.3.), and get a response to RETURN before continuing. If the circuit is operational, then exit from that @DOWN and continue with the remainder. Come back to the failed controller and try it again, individually (c.f., Section VI.2.4.1.).

If you receive a message of "TRAP ERROR" after entering the "GO" command, then the problem is the multibus backplane. Ask the workstation manager to reset the station and re-enter that station's boot instructions.

Exit each "@DOWN" in the ALLDOWN sequence by pressing

```
~.
```

and continue with the next station. DO NOT allocate the device before the ALLDOWN or @DOWN.

(6) Once all the NIPS have been started and all workstation managers report that their stations are operational and ready for connections, resume booting the network.

On DEMO, in the NETCMD window, start the network manager process: issue command

```
NETCMDW 200
```

Finally, submit the following script file names in response to the network manager's "Command:" prompt, in the order given.

```
@vwsbase
@hwsvall
@twsvall
@hvsbase
@IWSALL
@ALLDB
@CELLALL
@mhshvs
@mhshmb
```

(7) While NETCMD is processing the mailbox commands contained in the script files, you can go ahead and connect the CELL and MHS PCs to their common memory front end using their respective communications processor.

In the CELL window, issue commands:

```
su - wenger
cd cmcell/cell
cell
<and press an extra RETURN>
```

In the MHS window, issue commands:

```
su - wenger
cd cmcell/mhs
mhs
<and press an extra RETURN>
```

When these commands are completed, the CELL and MHS processes can be started on the respective PC.

(8) After the last script file has been processed (step 6) and the NETCMD "Command:" prompt reappears, and step 7 has been completed, the network is fully operational and ready to carry user data between systems.

2.3. <u>How To Verify That Network Circuits Are Operational</u>

This is only performed for serial RS232 and ethernet TCP/IP circuits that utilize the local broadband token bus network called AMRFnet.

2.3.1. Ethernet TCP/IP Circuits

Use the TCP/IP Telnet service to login to a remote computer host that accepts remote logins across the (TCP/IP) network path that you require to support your network configuration. If you are successful in your login, then the network path is operational. If you are unable to login, then it is possible that the AMRFnet Ethernet bridge is not operating properly: contact the AMRFnet network manager for further analysis and/or repair.

2.3.2. Serial RS232 Circuits

The operability of serial circuits across the AMRFnet can be established by determining whether their status is "SESSION IS ACTIVE" or "SESSION IS NOT ACTIVE". To do this, you must use a local asynchronous terminal and connect to a control port on the AMRFnet. Once connected, you must know the port number for the connection you are checking on. Table VI-3 lists all currently-assigned ports.

Table VI-3. Pertinent AMRFnet Port Assignments

Workstation	AMRFnet Port
ATC	013e01
HGP	0dd03
HMB	0dd04
HMC	0dd01
HRC	0dd02
HVS	0dd05
HWS	0dd00
IWS	01c01
TWS	013e00

The following procedure can be used to determine whether a station connection is ACTIVE. Press ENTER or RETURN at the end of each of your responses to menu prompts.

(1) Find a terminal that is connected to the AMRFnet (has the AMRFnet menu displayed).

(2) Press 'A' to make a connection.

(3) Press 'B' to connect by address (you will be prompted for an address).

(4) Enter '0609101' as the address to which you wish to be connected.

(5) When you see the "Connected" status message appear, press ENTER or RETURN several times. You should see several iterations of the "Invalid Choice" and "Selection:" messages from the AMRFnet. Press 'G' to perform network management functions.

(6) You will be prompted for a password. This password must be obtained from the AMRFnet network manager, and not disseminated to unauthorized individuals. Enter the password.

(7) You will now be prompted with "Enter Command:". Issue command "LIST xxx", where "xxx" is one of the addresses from Table 2, and the network will tell you the status of the connection. For example, "LIST 01c01" to determine the status of the connection for the IWS.

(8) If the status is "SESSION IS ACTIVE", then remedial action is unnecessary. Check any other connections you wish. Continue with step (10) when you're done.

(9) If the status is "SESSION IS NOT ACTIVE", then issue command "DISABLE nnn", where "nnn" is the name of the workstation. For example, "DISABLE IWS". Wait approximately 5 seconds, then issue command "ENABLE nnn". Wait approximately 5 seconds, then continue with step (7) to recheck the connection status. If you have done this several times without seeing status "SESSION IS ACTIVE", then contact the AMRFnet network manager for assistance.

(10) Once you have checked all network connections, exit from network manager mode by entering command "RETURN". This prohibits an unauthorized user from accessing the control functions.

(11) Disconnect your terminal from the remote control port by pressing the transition character to get back the local menu (usually, this is the BREAK key), then select the appropriate menu option to ABORT the connection ('E') and confirm the action by pressing 'Y' when requested.

2.4. <u>How to Start Individual NIPs</u>

2.4.1. Multibus Systems

All communication boards for the multibus-based systems have board RESET buttons installed in Control Room 2 that connect to the RESET function on the respective workstation. The boards occasionally are in a "strange" state when power is applied to the multibus units, or may occasionally require RESETs for other reasons. Whenever the term RESET is used in the following procedures, it refers to pressing the respective workstation's RESET button. The anticipated result in response to pressing the RESET button is an indication that the board has been reset. For the boards installed in the systems of Section VI.2.4.1.1. and Section VI.2.4.1.2., "MACSBUG ...." is displayed. For the boards installed in the systems of Section VI.2.4.1.3., "Pacific Micro ...." is displayed. Each of these displays is followed by the ">" prompt indicating that the board is ready for further instructions.

If you don't see the appropriate message in response to RESET, then:

(1) check the respective AMRFnet circuit and make sure that the session is active. TThe necessary steps are detailed in Section VI.2.3.2.

(2) if #1 doesn't work (session is already active), then press "~." to exit the DOWNload program and try the entire @DOWN procedure again. Sometimes, for reasons currently undetermined, this "trick" works.

2.4.1.1. HWS, HMC, HBD, HRC, HGP

On the terminal (or in the window) connected to the VAX, issue the following set of commands for each of the above-named workstations that is to be included in the configuration.

(1) @DOWN xnnn

where 'x' is mandatory, and 'nnn' is the 3-letter controller acronym. For example, 'xHRC' is the appropriate entry for the HRC. This command invokes a specially-written terminal communications interface to the VAX terminal port connected to the controller's serial interface. Do not ALLOCATE the port before executing this step: the DOWNload procedure will allocate and deallocate the port itself. This avoids "hanging" the port and inadvertently causing a NIP crash on the VAX.

(2) Press the respective controller's RESET button and wait for the "Macsbug ..." prompt.

(3) Enter the following commands in uppercase at the ">" prompt:

```
CAL FE2000
GO
```

(4) After you enter command "GO", you will see a repeating pattern of characters on the screen. Depending on whether you are using a VT100-type terminal or a window on DEMO, you will either hear the BELL or see the window go into reverse video momentarily approximately every 3 seconds. This is the normal behavior that you can expect to observe if the NIP has started properly.

If, instead, you get an error message of "TRAP ERROR", then there is a multibus bus error: ask the workstation operator to reset the controller, and resume with step 3, above, when the reset has been completed.

(5) After you have verified that the NIP is operating properly, enter

```
~.
```

to exit the DOWNload program. Resume with step (1), above if additional workstation controllers are to be included in the network configuration.

2.4.1.2. HVS

To start the NIP for the vision system, simply follow the same steps outlined in the previous section (VI.2.4.1.1.), however, substitute the following commands in step (3):

```
CAL FE2000
SR 2700
PC 1000
GO
```

2.4.1.3. TWS, ATC

On the terminal (or in the window) connected to the VAX, issue the following set of commands for each of the above-named workstations that is to be included in the configuration.

(1) @DOWN xnnn

where 'x' is mandatory, and 'nnn' is the 3-letter controller acronym. For example, 'xTWS' is the appropriate entry for the TWS. This command invokes a specially-written terminal communications interface to the VAX terminal port connected to the controller's serial interface. Do not ALLOCATE the port before executing this step: the DOWNload procedure will allocate and deallocate the port itself. This avoids "hanging" the port and inadvertently causing a NIP crash on the VAX.

(2) Press the respective controller's RESET button and wait for the "Pacific Micro ..." prompt.

(3) Enter the following commands (in upper or lowercase) at the ">" prompt:

```
g980000
r2
2700
r3
1000
go
```

(4) After you enter command "GO", you will see a repeating pattern of characters on the screen. Depending on whether you are using a VT100-type terminal or a window on DEMO, you will either hear the BELL or see the window go into reverse video momentarily approximately every 3 seconds. This is the normal behavior that you can expect to observe if the NIP has started properly.

If, instead, you get an error message of "TRAP ERROR", then there is a multibus bus error: ask the workstation operator to reset the controller, and resume with step 3, above, when the reset has been completed.

(5) After you have verified that the NIP is operating properly, enter

```
~.
```

to exit the DOWNload program. Resume with step (1), above, if additional workstation controllers are to be included in the network configuration.

2.4.2. Non-Multibus Systems

The network interface processes (NIPS) for all of the computers in the Inspection Workstation cluster (IWS, CMM, SRI, IRC) and the front end common memory computer, called DEMO, (for VWS, PPL, CDWS, CELL and MHS) will automatically be started when the station software is executed, and therefore require only some coordination between the network operator and the workstation operator to indicate that the workstation has been initialized before the network boot process can continue.

If any of the NIPs of the IWS cluster must be restarted, then the respective controller program must be restarted.

If the NIP on DEMO must be restarted, first make sure that the old NIP has been aborted (or abort it yourself). Then, use the mouse to call up the suntools menu, select the NETMGR submenu, and select the TCPNIP option to restart the NIP.

2.4.2.1. The VAX's TCPNIP

To execute TCPNIP without options enter:

```
RUN TCPNIP
```

TCPNIP can also be initiated with options.

d	turns on debug statements
t	specifies an alternative port number

In order to specify options when the program is initiated, a logical symbol must be defined:

```
TCPNIP :== $USER1[NETWORK.TVAXV1]TCPNIP
```

TCPNIP can then be started with or without options as follows:

```
TCPNIP
TCPNIP d
TCPNIP d t1590
```

2.5. <u>If a Restart is Necessary</u>

If it becomes necessary to restart the network configuration, all previously-connected NIPs must be RESET and restarted. Additionally, it is important to purge two common memories: the one on the VAX, and the one on DEMO.

The DEMO common memory is purged by aborting the common memory process in the CMM window and then restarting it. It is restarted by calling up the suntools menu, selecting the NETMGR submenu, and specifying the CMM option. Once it restarts, all client process (VWS, PPL, CDWS, CELL and MHS) must reconnect to it.

The VAX common memory is purged by aborting the VAX NIP(s), MBHAND, and any other process attached to it: notably, the IMDAS processes. Abort the network processes by

(1) exit NETCMD in the NETCMD window on DEMO with command 'q'

(2) on the terminal on which the TCPNIP is running, press control-C to abort the TCPNIP process

(3) on the same terminal, issue command NETKILL to terminate the serial NIP and MBHAND

(4) have the IMDAS operator exit all IMDAS operations

(5) after all of the above steps have been completed, restart the network in accordance with the instructions of Section VI.2.

If the common memory is not purged, it is very likely that the new NIPs will attempt to execute the command still remaining in the (old) common memory, and will result in a deadlocked status from which that NIP is unable to recover.

3. OPERATING THE NETWORK

3.1. <u>Managing Mailbox Connections</u>

3.1.1. Making (New) Mailbox Connections

The network architecture enables us to make mailbox connections dynamically; that is, while the network is operational. To make a connection, it is only necessary to insert an entry for the specific connection into the NIP's mail delivery table. The only stipulations are:

(1) Both ends (NIPs) of the connection must be operating.

(2) There must be room for the specified mailbox at each network node where the connections are to be made.

(3) There must be room in the NIP's mail delivery table for the entry of an additional connection.

Once all these stipulations are met, the connection can be made by submitting the appropriately-formulated command to NETCMD. Refer to Section III.2.4.3.1. for the command structure format.

Since the purpose of the connection is to transfer data FROM one mailbox TO another, you must connect the input side before the output side. As soon as the output connection is made, the network attempts to make a mailgram delivery. For example, the following two CONNECT commands first connect the input side and then the output side.

```
hmb ci h_mbd_cmd,    248  hws, 4FD1   040000
hws co h_mbd_cmd,    248  hmb, 4FD1   040400
```

3.1.2. Breaking Mailbox Connections

The network architecture enables us to break mailbox connections dynamically. That is, while the network is operational. To "break" a connection, it is only necessary to remove the appropriate entry from the NIP's mail delivery table. The only stipulations are:

(1) The NIP, at the node where the connection is to be broken, must be operating.

(2) The OUTPUT connection must be broken before the INPUT connection. This avoids having the NIP at the output node expending a lot of time attempting to deliver a mailgram that will not be accepted (or acknowledged).

NOTE: If the output NIP is no longer operational, then the OUTPUT connection should not be broken or it can cause the NIP to hang.

Once all these stipulations are met, the connection can be broken by submitting the appropriately-formulated command to NETCMD. Refer to Section III.2.4.3.1. for command structure format.

For example, the following two DISCONNECT commands first break the output side and then the input side.

```
hws do h_mbd_cmd,    248  hmb, 4FD1   040400
hmb di h_mbd_cmd,    248  hws, 4FD1   040000
```

3.2. Removing Stations From The Active Configuration

3.2.1. Station Is Active

To remove an active station from the active network configuration, all you need to do is to disconnect all of that station's mailbox connections. After the last mailbox connection has been disconnected, the workstation is no longer included in the active network configuration.

It is important to disconnect the mailboxes in the REVERSE ORDER in which they were originally established. That is, to disconnect the output side of a connection before disconnecting the input side. This, of course, must be done from the perspective of the station being removed.

EXAMPLE: The following commands connect the HWS to the HRC thru the VAX using the serial network links:

```
! Connect HWS NIP links
vax co hws_nip_cmd,    *68,1 hws, 001
vax ci hws_nip_sts,   *212,2 hws, 000
hws co hws_nip_sts,    212,2 vax, 000    0
! Connect HRC NIP links
vax co hrc_nip_cmd,    *68,1 hrc, 001
vax ci hrc_nip_sts,   *212,2 hrc, 000
hrc co hrc_nip_sts,    212,2 vax, 000    0
! Connect cmd and status mailbox links from HWS to HRC
vax ci h_rcs_cmd, *248  hws,  4FB1
hws co h_rcs_cmd,  248  vax,  4FB1 040200
hrc ci h_rcs_cmd, *248  vax,  4FB1 0F6000
vax co h_rcs_cmd, *248  hrc,  4FB1
vax ci h_rcs_sts, *248  hrc,  4BF1
hrc co h_rcs_sts, *248  vax,  4BF1 0F6200
hws ci h_rcs_sts,  248  vax,  4BF0 040300
vax co h_rcs_sts, *248  hws,  4BF0 0F6200
```

If we wanted to remove the HRC from this network configuration, we would submit all of its mailbox connection commands, in reverse order, as DISCONNECT commands, and carefully disconnect all OUTPUT mailboxes before disconnecting the INPUT mailboxes. The NIP command and status disconnects are performed last.

```
! Disconnect HRC-HWS status mailbox links
hrc do h_rcs_sts, *248  vax,  4BF1 0F6200
vax di h_rcs_sts, *248  hrc,  4BF1
vax do h_rcs_sts, *248  hws,  4BF0 0F6200
hws di h_rcs_sts,  248  vax,  4BF0 040300
! Disconnect HRC-HWS command mailbox links
hws do h_rcs_cmd,  248  vax,  4FB1 040200
vax di h_rcs_cmd, *248  hws,  4FB1
vax do h_rcs_cmd, *248  hrc,  4FB1
hrc di h_rcs_cmd, *248  vax,  4FB1 0F6000
! Disconnect HRC NIP links
hrc do hrc_nip_sts,    212,2 vax, 000    0
vax di hrc_nip_sts,   *212,2 hrc, 000
vax do hrc_nip_cmd,    *68,1 hrc, 001
```

3.2.2. Station Is No Longer Active (Crashed)

If a station crashes, it is only necessary to cancel the connections on the other stations still active. The same guidelines should be followed, as if the station were still active. That is, the (remaining) connections should be broken in the reverse order in which they were made.

Using the network configuration given at the start of the example of Section VI.3.2.1., if the HWS crashes, the resulting disconnect commands would be

```
! Disconnect HWS command & status mailbox links
vax do h_rcs_sts, *248  hws,  4BF0 0F6200
vax di h_rcs_cmd, *248  hws,  4FB1
! Disconnect HWS NIP links
vax di hws_nip_sts,   *212,2 hws, 000
vax do hws_nip_cmd,    *68,1 hws, 001
```

3.3. Inserting Stations Into The Active Configuration

3.3.1. New Stations

New stations can be added to the active configuration at any time by following these steps:

(1) Determine if the supporting network circuit already exists. That is, if you are going to add one of the IWS components (CMM, SRI, or IRC), then IWS must be active first; if you are going to add one of the clients of the front end common memory server (CELL, MHS, PPL, VWS, CDWS) to exchange data with the VAX or nodes that communicate through the VAX, then the TCP/IP Ethernet NIP must first by active. In other words, the supporting NIP(s) must be active.

If the supporting network circuit doesn't already exist, establish it first, in accordance with the directions of Section VI.2.

(2) Submit the mailbox connection commands to the network manager in the appropriate order. "xxxBase.net", if it exists, must precede any other connection commands. "xxx" identifies the workstation acronym: an example is TWSBase.net

[NOTE: PPL, VWS, CELL, MHS and CDWS all use the same "xxxBase.net" file, namely VWSBase.net. This basic script file must only be invoked once for the entire front ended group. If invoked more than once, then multiple NIP mail delivery table entries will be made, resulting in multiple deliveries of mailgrams. This is not necessarily fatal, but will burden the network with unnecessary traffic.]

3.3.2. Previously-Removed Stations

Stations that have been previously removed from the active network configuration, and have had all their mailboxes disconnected, are reinserted into the active network configuration by following the same procedure as for NEW stations.

3.3.3. Stations That Crashed And Were Rebooted

An attempt to reinsert a station whose mailboxes have not been totally disconnected can be very difficult or impossible, depending on the contents of those mailboxes: particularly the xxx_NIP_CMD output mailbox on the VAX. Likewise, it is often impossible to determine or recall which mailbox connections had been successfully established.

The recommended procedure is to:

(1) Disconnect all existing mailbox connections. No harm is done if you issue a mailbox DISCONNECT command to a NIP that currently doesn't have that connection in its mail delivery table.

(2) Restart the station as a NEW station inserted into the active network configuration.

3.4. Monitoring Operation Status

Refer to Section III.2.4.2. for detailed functional descriptions of particular NETCMD display fields. However, in order to monitor the status of any network connection, it is only necessary to watch the rows labeled as follows:

MDT Entries	- keeps track of the number of mailbox connections made for that station. This number should increase as a CONNECT command is processed, and decrease as a DISCONNECT command is processed.
TIME SINCE	- indicates how long ago since the NIP sent its status. The NIP is programmed to automatically report its status every 30 seconds. By pressing the RETURN key (with the Sun mouse arrow in the NETCMD window) at the "Command:" prompt several times, you will get immediate NETCMD display updates of this status field. Although these numbers will increase, they should eventually (within 1-2 minutes) return to 00 seconds. [NOTE: some stations, notably IRC, deny the NIP access to the CPU for periods of time longer than 30 seconds, so it is "normal" if the time exceeds 30 seconds, but abnormal if it approaches 5 minutes.]
NIP STATUS	- Indicates whether the NETCMD thinks the remote NIP is UP or DOWN. NETCMD changes the NIP's status to DOWN if it has not received a status report in more than 60 seconds, and changes it back to UP when a status report is received in response to a POLL or other command.

3.5. Configuration Shutdown

3.5.1. Orderly Shutdown

An orderly shutdown of the network requires that each individual mailbox connection established during network operation be disconnected (in reverse order) before workstation controllers are powered off. However, this is seldom the case in practice. The "panic" shutdown procedure is used, instead.

3.5.2. Panic Shutdown

The panic shutdown consists of:

(1) If the network is to be shutdown permanently, then power all the workstation controllers OFF.

Logout of the VAX (on both the VT100 and the NETCMD window), then logout of DEMO.

(2) If the network is to be shutdown for an immediate reboot, then purge common memory on DEMO and the VAX and restart all NIPs. (See Section VI.2.5.)

VII. LESSONS LEARNED

1. COMMON MEMORY

Libes [8] chronicles our experiences with common memory, and provides suggestions for a minimal implementation to guide future implementors. Summarizing, some things to avoid are statically-sized variables and typed variables. Additionally, there is a need to make the common memory interface in multi-user systems sufficiently robust to preclude the death of a single user process from blocking common memory access. (Such blocking is possible with the VAX common memory, but is precluded in the Sun common memory).

The concept of global common memory with transparent network services can create problems that arise from that transparency. If a problem occurs in the local common memory, the network, the remote common memory, or the remote process, it looks like a local common memory problem to the (local) user. This means that a large number of people can be involved in debugging a problem, and people have to be convinced that the problem actually lies with their part of the system. More flexible diagnostic tools are necessary to simplify this process.

2. NETWORK

2.1. I/O Using TCP Stream Sockets

2.1.1. Stream Sockets

The TCP version of the Sun NIP uses the stream type sockets provided by Berkeley UNIX. Stream sockets provide for the bi-directional, reliable, sequenced and unduplicated flow of data without record boundaries. Because record boundaries are not preserved across the network when using stream sockets, this must be done by the two communicating NIPs.

2.1.2. Connection Establishment

The establishment of TCP connections is asymmetric with one process acting as the server and the other process as the client. The algorithm for determining the role of each process in the connection process is described earlier (Section V.2.2). However, once the connection is established, the client/server model is no longer valid and the socket is treated as a symmetrical connection by the two NIPs.

2.1.3. Record Length And Stream Sockets

Since stream sockets do NOT preserve the record length of the write to the remote process, this is handled by the two cooperating NIPs. We used the sized_io routines from the stream library /usr/lib/libstream.a. These work quite simply by prefixing the record with the length in bytes of the record. That is, the format is

```
+-------+-------+-------+-------+-------+-------+-------+----
|       |       |       |       |       |       |       |
|         record size == N      |  record data ....
|          (4 bytes)            |  (N bytes)
|       |       |       |       |       |       |       |
+-------+-------+-------+-------+-------+-------+-------+----
```

This worked well until we discovered the write deadlock problem in using TCP stream sockets in blocking mode. We solved the problem by implementing the same record structure on non-blocking sockets (see below). The write deadlock problem also became apparent in the Sun common memory system sockets.

2.1.4. Socket Configuration

Upon opening a socket, it is configured for both asynchronous and non-blocking I/O. When in asynchronous I/O mode, the system delivers a system signal whenever new data arrives on the socket. In non-blocking mode, read and write calls on the socket never block but instead return errno=EWOULDBLOCK when no data is available on a read or the socket is full on a write.

Asynchronous mode is used so a system will deliver a signal (interrupt) when new data arrives at the socket. This is much preferred over continuously polling the socket for new data.

Non-blocking mode was used to avoid write deadlocks on the socket. Sockets are implemented with an internal limit of 4096 bytes. That is, if the number of unread bytes in the socket reaches 4096, the socket is full and further writes will block until there is space available in the internal socket buffers. Sockets are full duplex. Therefore if both processes simultaneously attempt to write to the socket more than a combined total of 4096 bytes, they will both block and not complete their writes (i.e., a deadlock).

This was all brought about by the fact that we could not implement the actual physical read of the socket in the SIGIO interrupt handler code. We could not do this since we first look at the size of the incoming packet via a read before getting the buffer from the buffer manager to read into. You cannot access the buffer pool in interrupt code since this runs the risk of confusing the free buffer list, etc. Therefore, we had to run the actual reads outside the interrupt code. Since we had to be able to read when the sockets got full (in order to avoid deadlocks) we had to make the sockets non-blocking.

As a final side effect we discovered that in non-blocking mode, output text length must not be too large (the limit is somewhere between 2000 and 3000 bytes, I suspect 2048). If it is too large, you get errno=EMSGSIZE and the packet is NOT sent. You cannot use ioctl(SIGCGHWAT) to see how much space there really is since it isn't supported yet in Version 3.0 of Sun Unix. Therefore large output is written a chunk at a time. I chose 1000 bytes as the chunk size. (TCPMSGSIZE == 1000 bytes)

All this was discovered late in the development, in retrospect things would have been implemented quite differently. For example, not getting the buffer in the actual read routine but instead somehow having one ready all the time would have eliminated the need for non-blocking I/O... or somehow allowing access to the buffer pool from interrupt code. Using a socket pair for communications would double the available buffer space before the deadlock occurs, but does not eliminate the need to perform the actual socket read in the interrupt handler routine.

Using two unidirectional sockets per connection would eliminate the need to use non-blocking I/O, and would also eliminate the socket full problem in both the NIP application of sockets and the SUN common memory system. Asynchronous mode would still be used so that the SIGIO signal would be delivered.

2.2. Network Manager Functions

The use of a human network manager was extremely beneficial during the development of the network services. However, as both the number of systems to connect and the number of interconnections have increased, it has become obvious that some additional method must also be developed. It is our intention to support user-directed connections, whereby a user process can send a command to its local NIP and request that a connection be established (or broken) with some remote system. This will necessitate making all local connection statically available to the user, and the network functions will no longer be totally transparent to the user process. These later services will be provided on all systems, and will not preclude the use of the current network management functions (NETCMD).

2.3. Computer-Dependent Byte Ordering

Libes [23] thoroughly discusses the problems encountered when transferring byte sequences between computers that have differing internal byte order sequences. In order to avoid data representation/interpretation errors, it is necessary to specify a standard byte sequence representation. All communicating computer systems, no matter what their internal byte sequencing, must convert to this standard representation before transferring data to another host, and must expect to receive data in the standard representation from other hosts. The International Standards Organization (ISO) has announced such a standard [24].

2.4. Commercially-Available Networking Products

Now that MAP and TOP networking products are becoming commercially available, it is our intention to begin their phased integration into the AMRF for evaluation. We expect that, eventually, all of our locally-developed networking software will be replaced with commercial products. Only common memory will remain, and we will have to develop new software to link it to the new underlying network.

Appendix A

AMRF Interprocess Communication in Multibus Systems

This appendix describes the standard implementation of AMRF mailboxes for multimaster (IEEE 796) Multibus-based microcomputer systems, using standard bus controls and having at least one bus-accessible (common) memory area.

It is of particular note that the NBS Robot Control System, in its current implementation, uses an additional access control protocol and does not fall under the provisions of this document.

Section references herein are to the main document - AMRF Interprocess Communication Standards, unless they begin with the letter A.

A.1. REPRESENTATION OF MAILBOXES

Multibus mailboxes follow the structure and rules for mailbox management described in Section III.1.

A.1.1. Mailbox Structure

The form of the mailgram is as specified in Section III.1.2. The mailbox contains an 8-byte header to support variable-length mailgrams, to provide change indication and to allow arbitration of simultaneous accesses to the mailgram data in a multiprocessor environment. The access control header has the following form:

Byte	1-2	3-4	5-6	7-8
	Write Lock	Read Lock	Sequence	Length

where:

Write_lock is a 2-byte integer containing either a 1 (locked) or a 0 (unlocked), set and cleared by the process which writes in that mailbox.

Read_lock is a 2-byte integer containing the number of processes currently engaged in reading information from that mailbox.

Sequence is a 2-byte integer which is changed by the writer to indicate an update to the contents.

Length is a 2-byte integer giving the length of the current mailgram in the mailbox.

The use of the access control header is explained in section A.2.

A.1.2. Mailgram Flow

Since the Multibus environments will almost universally be multiprocessor systems, interchanges between processes must comply with the guidelines on flow control specified in Section III.1.1.4.

In this environment, no synchronization can be expected between the sender and the receiver(s) of any mailbox for which flow control is not applied. Receivers will, of course, always get the current mailgram in the mailbox, but they may read the same one several times (because there is no guarantee that the sender on another processor will have completed an update cycle in the interim), or they may miss several intervening mailgrams (because the sender may have completed more than one update cycle while the receiver was busy).

Even when the cycle times of the sender and receiver are known to have a one-to-one or one-to-n relationship, but the sender and receiver are on different processors, the physical flow of the mailgrams is somewhat unpredictable. What occurs in these cases is known as a "race condition": whether the interchange works as expected depends on whether processor A gets to a particular instruction or memory cell before processor B does.

A.1.3. Mailbox Data Representation

The Intel 8086/8088 stores integers in binary, two's complement, least significant byte first.

The Motorola 68000 stores integers in binary, two's complement, most significant byte first internally, but the bytes may be inverted (that is, least significant byte first) on presentation to the Multibus.

Character strings are 7-bit ASCII with the high-order bit being zero.

The communications software is aware of the byte order of the originating processor for the header fields, and stores values compatible with that byte-order. It is not aware of the areas in which byte-order is significant within the mailgram, so that mailgram text areas are required to obey other AMRF conventions. At this writing, the AMRF standard is most significant byte first for integers.

A.2. ACCESS

A.2.1. Connection

Currently all mailboxes are "given", that is, they are preconstructed and preallocated for the control and data management processes. Control processes, therefore, associate an address (in bus-accessible memory) with each mailbox structure and reference it directly while following the protocols described below.

The communications system at this time has a "given" network mailbox table, specifying which local mailboxes should be transmitted over the network, and to where, and which local mailboxes should receive incoming network mailgrams.

A.2.2. Sending Mail

Sending mail is conceptually just a matter of storing into the various fields of a particular given mailbox. In order to assure integrity of a mailgram, however, one must guarantee that no process reads the mailgram while the sender is modifying it. This is the purpose of the "lock" words.

The standard write-access algorithm (expressed in Pascal) is:

```
mailbox.write_lock := 1;
while mailbox.read_lock > 0 do wait;
      {modify mailbox contents}
        . . .
mailbox.write_lock := 0;
```

If the processor (not the control process, but the CPU it uses) can afford to wait for all of the receiver processes to finish, then "wait" is no-operation, i.e. the processor loops testing the read_lock. If the processor time is needed elsewhere, in particular, if it is possible for a receiver running on the same processor to have been interrupted while operating on the mailbox, hanging the CPU in a loop is unacceptable. In this case, "wait" becomes the appropriate system call to relinquish the CPU.

A.2.3. Receiving Mail

Receiving mail is conceptually just a matter of fetching from the various fields of a particular given mailbox. In order to assure integrity of a mailgram, however, one must guarantee that the sender is not in the process of modifying the mailgram when this reader retrieves it. This is the purpose of the "lock" words.

The standard read-access algorithm (expressed in Pascal) is:

```
 repeat
   mailbox.read_lock := mailbox.read_lock + 1;
   v := mailbox.write_lock;
   if v = 1 then begin
       mailbox.read_lock := mailbox.read_lock - 1;
       wait;
       end {if};
 until v = 0;
 {fetch fields from mailbox}
  . . .
 mailbox.read_lock := mailbox.read_lock - 1;
```

Notes:

1. Incrementing and decrementing the read_lock are somewhat sensitive operations in multiprocessor systems. Unless all of the receiving processes are on the same processor (which is true in the case of only one receiver), the memory cell being incremented (or decremented) must be locked against intrusion by another processor while the read/modify/write memory cycle(s) of the incrementation occur.

   If the incrementation takes more than one cycle and there is no such protection, the following may occur:
   1. Processor A fetches the read_lock, currently zero.
   2. While processor A is incrementing the value to one, processor B fetches the read_lock, still zero.
   3. Processor A replaces the read_lock, now one, while processor B is incrementing its copy from zero to one.
   4. Processor B replaces the read_lock, again one.

   The result is that although two processes are actively reading the mailgram, the read_lock only shows one. When either process finishes and decrements the read_lock, the value will be a zero and a write may occur, even though the other process is still reading.

   By comparison, if there is only one processor (and one instruction), or processor A can lock the Multibus when it fetches and processor B must use the Multibus to access the memory cell, step 2 cannot occur - processor B cannot fetch the read_lock while processor A is incrementing it; processor B cannot fetch the read_lock until processor A replaces the incremented value and unlocks the bus.

2. If the processor (not the control process, but the CPU itself) can afford to wait for the sending process to finish, then "wait" is no-operation, i.e. the processor loops testing the write_lock. If the processor time is needed elsewhere, in

particular, if it is possible for a sender running on the same processor to have been interrupted while operating on the mailbox, hanging the CPU in a loop is unacceptable. In this case, "wait" becomes the appropriate system call to relinquish the CPU. In many cases, the "wait" may be just a "return", allowing the old value of the mailgram to be used.

Appendix B

Error Conditions And Messages

The following sections identify error messages that the operator or network manager may encounter while running the communications network. Error messages that are used for debugging purposes are not identified.

B.1. NETWORK MANAGER (NETCMD)

minimum width xxx
An attempt was made (when starting NETCMD) to set the screen width less than the minimum screen width

error from NIP detected
command file aborted
last line executed: xxx
The NIP (network interface process) has returned an error to the network manager in response to a command issued from a network script file. Further processing from the script file has been aborted, and the file has been closed. The last line executed was "xxx". The actual NIP error is displayed as the status on the "cmd #/status" line of the network manager display (Figure IV-1). The NIP errors are identified in Section B.2.1, below.

! xxxxxx xx xxxx
NETCMD echoes any command it encounters (entered at the keyboard or read from a network script file) that starts with an exclamation point.

can't read files recursively!
An attempt has been made to access a second network script file from within the current network script file.

unknown command: xxx
<self-explanatory>

abandoning command file due to error
<self-explanatory> The error that resulted in this message will have been identified by other diagnostic messages.

bad direction: xxx
The "direction" specified in command "xxx" must be Input, Output, or Duplex.

bad mailbox name: xxx
<self-explanatory>

no length: xxx
: Command "xxx" does not include a mailbox length specification.

bad mailbox length: xxx
: Command "xxx" has an invalid mailbox length specification.

no station: xxx
: Command "xxx" is missing a station name. This may either be the source or the destination name, or both.

bad syntax identifier: xxx
: Command "xxx" does not parse properly. Check that it is in the correct format.

no address: xxx
: Command "xxx" does not contain an address field entry. This should be the actual memory address for the Multibus-based systems, and zero (0) for all others. Numbers are hexadecimal.

bad address: xxx
: Command "xxx" does not contain a valid address. This should be the actual memory address for the Multibus-based systems, and zero (0) for all others. Numbers are hexadecimal.

B.2. NETWORK INTERFACE PROCESS (NIP) MESSAGES

B.2.1. General Observations

With the exception of the network interface processes (NIPs) resident on the AMRF VAX computer, the NIPs do not perform any activity logging. The VAX NIPs generate a line of output for each mailbox transaction. This information is used for debugging purposes, and can either be displayed on a terminal screen during the operation of the network, or sent to a logging file for later examination.

Each NIP has some additional instructions coded in to support enhanced diagnostics and operation monitoring. The execution of these instructions is effected by either specifying an argument on the command line that starts the NIP, or by setting an internal flag and recompiling and relinking the source code. The significance of these debugging messages is specific to the section of program code involved, and is not necessarily relative to the overall network architecture. These messages are not described herein.

There is a set of error messages common to all NIP's. These are identified and described in the next section.

B.2.2. Messages Common To All NIP's

In general, the NIP will not attempt to display an error message at the host system on which it is operating. Instead, the NIP will return a status code in its status mailbox (xxx_NIP_STS). The following messages are displayed in the status field of the "cmd #/status" line of the network manager display (Figure IV-1) to report the NIP status.

The numeric codes that are displayed in this manner identify errors that may have occurred at any level of the network model. Each error is displayed as a four digit hexadecimal code, and the leftmost digit specifically identifies network layer, as

0x1000	physical layer (device) errors
0x2000	link layer errors
0x3000	network layer errors
0x4000	transport layer errors
0x5000	session layer errors
0x7000	application layer errors

General guidelines for the interpretation of error codes:

odd	unusual (but normal) occurrence at xxx level
even	real error at xxx level
0x00	out of space in the xxx control tables
0x10	no find in the xxx control tables
0x01	normal completion state to be reported up from xxx
0x0F	action deleted state reported up from xxx

Link layer error codes:

0x2000	no link control block available
0x2010	link is not open
0x2020	unrecoverable checksum errors
0x2030	no buffer for receive
0x2019	link is disconnected
0x2801	output completed on buffer
0x280F	output cancelled on buffer
0x2803	input completion

Network layer error codes:

0x3000	no network control block available
0x3010	site name not in network table
0x3030	no distribution control block available

Transport layer error codes:

0x4000	=	no transport control block available
0x4010	=	no transport connection open
0x4801	=	packet acknowledged
0x480F	=	packet never acknowledged

Session layer error codes:

0x5000	=	no MDT entry available
0x5010	=	no active session with matching MDT entry
0x5020	=	unknown syntax identifier
0x5040	=	illegal size
0x5011	=	transport connected
0x5019	=	transport disconnected

Application layer error codes:

0x7000	=	no such command

B.2.3. Messages Common To TCP/IP NIP's

The TCP/IP NIPs have additional diagnostic code within them to report TCP/IP error conditions that are not shared by the other NIP's. The specific diagnostic messages will not be detailed here, since they deal with specific TCP/IP rather than operational or architectural errors. In general, the significance of the error will be immediately obvious (eg, "Unable to connect to host xxx"). The meaning of more obscure errors is detailed in the appropriate TCP/IP reference manual. This manual is specific to the respective host and TCP/IP vendor. For example, the AMRF Suns use Sun Microsystems TCP/IP product, so the appropriate Sun documentation should be referred to. On the other hand, the AMRF VAX uses The Wollongong Group WIN/TCP product, and its reference material would be appropriate for TCP/IP error messages generated by the VAX's TCP NIP.

B.3. COMMON MEMORY ERROR MESSAGES

B.3.1. General Comment About Common Memory Messages

There are no common memory error or diagnostic messages for any implementations except for the Sun and VAX. The VAX common memory implementation has been described completely [14,15], and will not be repeated here. The Sun common memory errors are identified and described below.

B.3.2. Sun Common Memory Errors

Most types of errors are reported at the user program. Some messages cannot be reported back to the user, and are reported at the common memory process itself. Some errors are serious enough that they are reported at both the user and common memory process.

Most user errors can be fixed when identified. For example, writing into a variable declared to be read-only would be a user-error.

Since user errors indicate a user-programming error, the common memory system usually prints out a message indicating the problem. It also returns an error code if possible. It is sometimes not possible to do this. For instance, the above example would not be detected until after cm_set_value returned. The actual message would be printed by cm_sync when it is processing incoming messages from the common memory manager. Most types of errors are detected by cm_sync.

System errors are caused by limitations in the common memory system itself, the environment it is running in and the user demands upon the system. Often, these cannot be avoided. For example, if the user attempts to send too much data to the common memory at once, the maximum message size can be exceeded.

In order to make it easy for the user to identify the error and its associated corrective action, the following section lists all known Sun common memory error conditions, their associated message, and corrective action.

B.3.2.1. Listing of Error Messages

cm_init:

returns E_CM_INIT_FAILED
initport(client): Connection refused
Problem: common memory manager is not running.

cm_sync:

failed to send msg to common memory manager. Common memory manager disappeared?
Problem: common memory manager died. Detected while writing to it.

cm library (version #) is older/newer than common memory manager (version #)
Problem: common memory manager is a different version than the libraries your code is compiled with. This can also be caused by a corrupted message. The is usually identifiable by wildly different version #s.

bad slot encountered...aborting msg
user_decode_slot: unknown slot type (#)...msg aborted
Problem: corrupted message or internal error in common memory system.

Common memory manager: error processing variable <name> - error message
Problem: common memory manager detected error "error message" in processing "name". See below.

get_slot_read_response: <name> unknown (sent from common memory manager)
Problem: corrupted message or internal error in common memory system

too much data for msg!!
output msg size = # slotsize = #
Problem: User value is too large for common memory system configuration. Either user error, or message size limit should be increased.

cm_sd_free() called on nonmallocable object?
Problem: internal error in common memory system

*:

error: bcopy src/dest is null ptr
Problem: internal error or user error. If user error, check elements of cm_value structures to see that they are consistent.

common memory manager:

bind() failed
initport(server): Address already in use
failed to initialize connection socket
Problem: another common memory manager is running, or a process already has the common memory manager connection socket open.

get_variables(name) failed
Problem: too many variables in common memory manager.

process <name> is being antisocial on fd #
Problem: process has requested wakeup service but is not listening to common memory manager updates.

slot bad
Problem: corrupted message or internal error in common memory system

slot error in <name> type # - error message
Problem: corrupted message or internal error in common memory system or user error. See error message. This message is sent back to the user. See below.

Error messages generated by the common memory manager and sent back to the user:

version
Problem: version mismatch. See above.

bad slot type
Problem: corrupted message or internal common memory system error.

not enough common memory to declare variable
Problem: too many variables stored at common memory manager.

cannot get nonexclusive write access
Problem: a process has already received exclusive write access to this variable.

undeclare of undeclared variable
Problem: a nonexistent variable is being undeclared.

variable has not been declared
Problem: attempt to read/write variable not yet declared.

not declared as writer
Problem: attempt to write variable declared as read-only.

get_slot_write: cm_flat_to_sd() failed! no space?
Problem: common memory manager ran out of memory trying to read user message. Indicates lack of system resources or user sent value that was too large.

```
        not declared as reader:
        problem: attempt to read variable declared as write-only.
```

There are several places in the common memory system where memory is dynamically allocated. These may fail with an error such as

```
        func: failed malloc(object,size)
```

or

```
        resized failed! - out of space
```

where "func" is the Common memory system function calling malloc, "object" is the object being malloc'd and size is the size of the object.

These errors typically indicate that either:

1) the user is storing or receiving incredibly lengthy values (probably by mistake), or

2) the system is running out of internal space

Appendix C

Mailbox Label Assignment

Conventions for the Network Session (Mailbox) Labels in the interim AMRF network are as follows:

Command form:

```
    C<dir> <mailbox-name>,<length>  <station>,<label>  <address>
```

The station names are: VAX, HWS, ATS, HMC, HRC, etc

The <label> is three or four hex digits, specifying:
    [workstation], source-process, destination-process, and function respectively.

Process identifiers are:

0 =	Network (NIP)
1 =	Data Manager
2 =	Cell
3 =	Material Handling Workstation
4 =	Horizontal Workstation
5 =	Turning Workstation
6 =	Inspection Workstation
7 =	Vertical Workstation
8 =	Cleaning & Deburring Workstation
9 =	undefined
A =	machine tool control
B =	robot control system
C =	second RCS (or gripper)
D =	material buffering
E =	vision/sensors
F =	workstation control

Function codes are:

0 =	Status
1 =	Command
2 =	Data
3-F =	undefined

Appendix D

Interface Specifications

D.1. COMMON MEMORY

D.1.1. Systems With Fixed Memory Allocation

The multibus-based implementations of common memory depend upon accessing the common memory location by address, rather than by a mailbox name. Since these implementations do not support dynamic memory allocation, each common memory mailbox has its address, size, and name predefined. This information is advertised to other processes and processors within the multibus environment in order to avoid unintentional reassignment or reuse of mailbox memory areas. Consequently, these common memory areas always exist, and only their (network) links are created dynamically.

Remote links to these common memory areas are created through a dialogue between the network interface process (NIP) in the multibus system and network manager located on another computer system and using a separate NIP, as described in Section III.2.4. Once the links are established, the mailgrams can be propagated to other local or remote mailboxes.

D.1.2. Systems With Dynamic Memory Allocation

The VAX, Sun, and HP implementations of common memory take advantage of their host operating system support for the dynamic allocation of memory. The mailbox (memory allocation) is made at the time the mailbox is declared, and dissolved (memory deallocated) at the time the mailbox is undeclared.

Appendix E

List of All NETCMD Script File Names

```
ALLDB.NET;2
ATCBASE.NET;2
ATCDB.NET;3
CELLALL.NET;1
CELLDB.NET;1
CELLDWS.NET;2
CELLHWS.NET;7
CELLIWS.NET;9
CELLMHS.NET;1
CELLTWS.NET;10
CELLVWS.NET;2
HGPBASE.NET;2
HMBBASE.NET;3
HMBDB.NET;9
HMCBASE.NET;2
HMCDB.NET;2
HRCBASE.NET;2
HRCDB.NET;11
HRCHGPV.NET;2
HVSBASE.NET;4
HVSDB.NET;11
HWSALL.NET;5
HWSBASE.NET;2
HWSDB.NET;5
HWSHMBV.NET;1
HWSHMCV.NET;1
HWSHRCV.NET;2
HWSVALL.NET;2
IWS.NET;1
IWSALL.NET;3
IWSBASE.NET;18
IWSCMM.NET;10
IWSDB.NET;9
IWSIRC.NET;2
IWSSRI.NET;14
MHSHMB.NET;2
MHSHVS.NET;3
PPLDB.NET;10
TWSATCV.NET;5
TWSBASE.NET;2
TWSDB.NET;13
TWSVALL.NET;2
VWSBASE.NET;2
VWSDB.NET;3
```

Appendix F

Network Hardware and Software Components

The following sections identify the hardware and software installed to support networking in each of the major processor categories: VAX, Sun, Hewlett-Packard, and Multibus.

F.1. MULTIBUS HARDWARE CONFIGURATION

F.1.1. Components of the Horizontal Workstation

The Horizontal Workstation is composed of the HWS, HMC, HRC, HGP, HBD, and the HVS. (The HVS also provides vision support to the Turning Workstation.) Each of these components has network service provided at two levels: a serial connection to the VAX, and an Ethernet connection to other nodes within the Horizontal Workstation.

The hardware to support these services is composed of an OB68K1A single board computer from Omnibyte Corp. and an EXOS/101 Ethernet controller card from Excelan, Inc. The OB68K1A CPU board is built around a Motorola MC68000 and is used for all protocol handling above the link layer. The CPU board also provides the RS232C ports for the serial links to the VAX.

The Excelan board provides a link layer interface to its host (the OB68K) across the Multibus backplane.

The OB68K1A comes with the MACSBUG monitor program. The monitor is used to interact with the network interface process (NIP) on a "debug" level during development. During standard operations, the monitor is simply used to start the NIP. The release level of the monitor is MACSBUG (OB68KMACS) 1.32.

F.1.2. Components of the Turning Workstation

The Turning Workstation is composed of the TWS and the ATC, with vision input from the HVS. Both the TWS and the ATC have serial service to the VAX. (There is no Ethernet communication between the component systems of the Turning Workstation.) The single board computer used to provide this service is a PM68D from Pacific Micro Computers. Like the OB68K, the PM68D is based on MC68000 but it has the capability to support RS449 high speed serial links (which may be used in the future).

The PM68D comes with the Prom monitor program. The monitor is used to interact with the network interface process (NIP) on a "debug" level during development. During standard operations, the monitor is simply used to start the NIP. The release level of the monitor is Prom Monitor Version 1.5/1.

F.2. THE VAX COMPUTER SYSTEM

The AMRF VAX operates with the VMS Operating System (v4.5). No special alterations were made to the operating system or the hardware in order to accommodate the network programs (NETCMD and NIP's).

The VAX uses both Ethernet and serial RS232 NIP communications. The serial ports used by the serial NIP are configured with the following parameters:

```
 Terminal: xxxxx:    Device_Type: Unknown     Owner: No Owner

   Input:   9600    LFfill:  0    Width: 132    Parity: None
   Output:  9600    CRfill:  0    Page:   24

 Terminal Characteristics:
   Interactive          No Echo
   No Hostsync          TTsync
   No Wrap              Scope
   No Broadcast         No Readsync
   No Modem             No Local_echo
   No Brdcstmbx         DMA
   No Line Editing      Overstrike editing
   No Secure server     No Disconnect
   No SIXEL Graphics    No Soft Characters
   No ANSI_CRT          No Regis
   No Edit_mode         No DEC_CRT

   No Typeahead         No Escape
   Lowercase            No Tab
   No Remote            No Eightbit
   No Form              Fulldup
   No Autobaud          No Hangup
   Altypeahd            Set_speed
   No Fallback          No Dialup
   No Pasthru           No Syspassword
   No Printer Port      Numeric Keypad
   No Block_mode        No Advanced_video
   No DEC_CRT2
```

The Ethernet communications are supported via an Ethernet communications controller obtained from Micom-Interlan and the TCP/IP software suite obtained from The Wollongong Group (we are currently using version 3.0).

F.3. THE SUN (DEMO)

All Ethernet-based communications used in support of the NIP are entirely derived from the hardware and software bundled with each

Sun. No special configuration changes were made to support the NIP, nor was any additional software purchased.

F.4. THE INSPECTION WORKSTATION

The Inspection Workstation uses several Hewlett-Packard 9000 Series 200 (HP 9000) microprocessors. The network interface process (NIP) utilizes a RS232 serial interface for network communications.

In order to offload interrupt processing from the main processor, and to enable an eventual programming change that would result in most network software being resident on the interface card, a model 98691A Programmable Datacomm Interface was installed in each HP 9000.

Special software was developed for the Z80 processor on the board to perform the necessary I/O functions and perform data transfers between the HP 9000 memory and the Z80 memory. A special software interface (ACIDVR) was written for the HP 9000 to facilitate communications between the two processors.

Appendix G

Local Transport Protocol

The transport protocol implemented in the AMRF is derived from the ANSI/ISO High Level Data Link Control Procedure (HDLC) [4], although the framing conventions and integrity checking procedures are deleted, because they are deemed proper to the data link layer (from which the protocol comes) but not to the transport layer. In addition, a segmentation service has been added, in order to accommodate large information units with limited packet sizes.

G.1 TRANSPORT LAYER SERVICE INTERFACES

The transport layer expects to provide services to some "upper" layer of communications activity, which we designate as the "user". It also expects to use a "lower" layer of communications services to accomplish the actual delivery of data units to a remote station. At the remote station, the transport layer expects to communicate with a "peer" transport which implements the protocol herein described.

G.1.1 User Interface

The transport layer accepts the following requests from its users:

a) Connect request: a request to open a new connection to some remote host.
b) Disconnect request: a request to break an existing open connection;
c) Data request: a request to send a data unit on an open connection.

It provides the following "indications" to its users, through a procedure designated by the user as its "service access point":

a) Connection confirm: a request to open a connection has completed, successfully or unsuccessfully;
b) Disconnect confirm: a request to break a connection has completed;
c) Disconnect indication: the remote station has requested that the connection be broken;
d) Data indication: a data unit has been received on an open connection;
e) Data confirm: a transmitted data unit has been acknowledged by the remote host;
f) Abort indication: a formerly open connection has been broken by the local transport without request, usually because of an underlying problem or nonresponsiveness of the remote host.

G.1.2 Network Interface

The transport layer expects the following services from the "network" layer:

a) Connect request: transport presents a request to open a link to some remote host;
b) Connect confirm: network reports successful completion or failure of a connect request;
c) Disconnect request: transport presents a request to close an open link;
d) Disconnect confirm: network reports completion of a disconnect request;
e) Data request: transport presents a data unit for transmission on an open link;
f) Data indication: network presents a data unit received on an open link;
g) Abort indication: network reports unrequested closure of a formerly open link, because of remote action or failure of an underlying service.

The existing transport implementation also uses the following service provided by the network layer implementation:

h) Data confirm: network reports successful transmission or transmission abort for every data unit presented in a data request.

Note that the implementation of connect/disconnect in the network layer may be nil or may involve some link layer activity, depending on the nature of the physical connection.

G.2. ELEMENTS OF THE PROTOCOL

The term "protocol data unit (pdu)" comes from the OSI model and means a string of bits which, taken together, form the basic unit of communication between peer services in a given layer of the model. Each transport pdu in the AMRF transport protocol comprises a type byte, a segmentation byte and an optional text data unit. In discussion of the protocol, pdu types are identified by their mnemonic code. Details of the representation are found in section G.3.8.

There are three general classes of pdus which can be transmitted: information pdus (I-pdus), which contain data, supervisory pdus (S-pdus), which control the transfers, and unnumbered pdus (U-pdus), which are used to initialize and shutdown the connection.

PDUs are further divided into "commands" and "responses". Information pdus are always commands; supervisory pdus can be either - a bit in the type byte determines whether a given S-pdu is a command or a response. U-pdus have individual types, and each U-pdu type is either always a command or always a response.

The two stations implementing a connection are considered equals (i.e., there is no master/slave relationship). Both stations can send all types of commands and each is required to respond to commands issued by the other station.

G.3. DETAILS OF THE PROTOCOL

G.3.1. Initial Connection

Initially, all connections are in the "disconnected state": a physical connection exists, but the network link is logically uninitialized. If a SABM pdu is received on a connection in the disconnected state, the host responds with a UA and places the connection in the normal operation state. If any other pdu is received on a connection in the disconnected state, the host responds with a DM and leaves the connection in the disconnected state.

When the "user" requests a connection to a particular remote host, the connection request is passed from the transport layer to the network layer, so that whatever connection activity is required for the designated host can be initiated. Then the connection to that host enters the "initial connection" state.

When a connection is in the "initial connection" state, the host transmits a SABM and waits for the receipt of a UA. When it receives the UA, it informs the user of the connection completion and goes to the normal operation state.

If a host receives a SABM on a connection in initial connection state, it transmits a UA, informs the user of connection completion, and enters the normal operation state.

If a host receives a DM on a connection in an initial connection state, the host must assume that the connection request is refused, report the failure to the user, and return the connection to the disconnected state.

If in this state some other pdu is received, the host transmits a DM on the connection, followed by retransmitting the SABM.

If in this state the connection times out, having received no UA, the host retransmits the SABM and waits to receive a UA. After the maximum number of retransmissions without receipt of a UA, the host aborts the connection attempt, sends the user an abort indication, and returns the connection to the disconnected state.

G.3.2. Data Transmission

When a user presents a data request on an open connection, the transport service queues the requested "service data unit" (sdu) for transmission to the designated host. If the length of the sdu is within the maximum transport packet size, the sdu is copied into a single packet buffer with the EOM (end of message) bit set to ONE and the segment number set to zero. Otherwise, the sdu is partitioned into several packet buffers, of which the first has segment number zero, the second has segment number one, and so on, and all but the last have the EOM bit set to ZERO. All packet buffers for the sdu are then queued for output (become "data waiting") in the order of construction.

When a connection is in normal operation state, the host may transfer information pdus whenever it has "data waiting". Each I-pdu corresponds to a single packet buffer and the segment byte of the I-pdu contains the segment number and EOM flag from the packet buffer. In addition, in the type byte, every I-pdu contains a sequence number (called N(S) in the representation). I-pdus are numbered sequentially on each connection, beginning with zero and repeating modulo 8, i.e. the sequence number after 7 is 0. Each I-pdu must be acknowledged by the receiver; the sender does not treat the transmission as complete until the remote host has acknowledged it (See G.3.4). A host may transmit up to 7 I-pdus before receiving acknowledgment. The limit of seven prevents ambiguity in the sequence numbers of outstanding I-pdus. A given implementation may elect to require acknowledgment after a much smaller group of I-pdus has been transmitted.

Requiring acknowledgment, also referred to as "polling", uses a flag contained in every command pdu, called the P-bit. Normally the P-bit is ZERO on I-pdus, and is set to ONE only when the transmitting host is demanding acknowledgment of this pdu (and all preceding pdus) before it can continue transmitting. Once a pdu is sent with P=1, the connection enters the "polling" state, and no other command pdu can be sent until the outstanding poll is cleared, i.e. until the connection re-enters the "normal operation" state. The transport may send response pdus while in the polling state; in fact, it may be required by the protocol to send response pdus in this state.

An outstanding poll is cleared, and the connection re-enters normal operation state, when the polling host receives any U-pdu (which will usually result in an immediate transition to some other state) or a response S-pdu with the F-bit set to ONE. If a poll is not cleared in some fixed length of time, it is said to "time out", and a timeout state is entered.

G.3.3. Data Reception

A receiving host examines each incoming pdu to verify that the pdu has a known pdu type. If the pdu is of an unknown type, it is erroneous and the receiving host simply discards it. Once the type is determined, the action taken depends on the type.

G.3.3.1 Information PDUs

If the pdu is an I-pdu, the type byte is processed for acknowledgment (see G.3.4). Then the sequence number is examined to verify that the pdu is in sequence, i.e. that N(S) matches the current "expected sequence number". If the pdu is out of sequence, or the receiver is "not ready" (see G.3.5), the pdu is discarded at this point.

If the pdu is in sequence, the I-pdu is "accepted". The segment byte is then examined and the following dispositions are possible:

a) the EOM bit is ONE and the segment number is zero:
In this case, the text unit is a complete sdu; the text unit is presented to the user as an incoming data indication.

b) the EOM bit is ZERO and the segment number is zero:
In this case, the text unit is the beginning of an incomplete sdu. An sdu assembly is begun and the text unit becomes the beginning of the sdu.

c) the EOM bit is ZERO and the segment number is not zero:
In this case, the text unit is the continuation of an incomplete sdu. The text unit is appended to the sdu being assembled.

d) the EOM bit is ONE and the segment number is not zero:
In this case, the text unit is the end of segmented sdu. The text unit is appended to the sdu being assembled, the assembly is completed, and the sdu is presented to the user as an incoming data indication.

When an I-pdu is accepted by the receiver, it must be acknowledged. The current "expected sequence number" is replaced by the sequence number of the new I-pdu incremented by one: N(S) + 1. If the receiving host has data waiting on this connection, then the new expected sequence number goes into the N(R) of the next outbound I-pdu; otherwise the host must send a response S-pdu with the new expected sequence number in N(R) (see G.3.5). In either case, the transmission of the new expected sequence number constitutes acknowledgment (See G.3.4). It is not necessary to send an acknowledgment for each I-pdu received - acknowledging the most recent I-pdu at the first opportunity implicitly acknowledges all of its predecessors.

G.3.3.2 Supervisory PDUs

When a S-pdu is received, the type byte is processed for acknowledgment (see G.3.4). Then the ready status of the transmitter is set from the Ready/Not-Ready bit (see G.3.5) and the pdu is discarded.

G.3.3.3 Unnumbered (Control) PDUs

When a SABM is received on a connection which is in normal operation state, the host interprets this as a "reset", sets N(S) and the next expected sequence number to zero, requeues any unacknowledged transmissions, transmits a UA, and enters the normal operation state.

When a SABM is received in any other state, the receiving host (re)initializes the connection as described in section G.3.1.

When a DISC is received, the receiving host breaks the connection as described in section G.3.7.

When a UA is received in a connecting or disconnecting state, the action taken is described in sections G.3.1 and G.3.7. When a UA is received in a normal operation state, the UA is ignored.

When a DM is received on a connection in any state, it is an indication that the remote half of the connection is not open. If it is received in a disconnected or disconnecting state, it is ignored. If it is received in an initial connection state, the action taken is described in section G.3.1. If it is received in any other state, the host must present an abort indication to the user and close the connection.

G.3.4. Acknowledgment

The type byte of an I-pdu or an S-pdu contains the P or F bit and the then-current value of the remote host's expected sequence number, hereafter referred to as N(R).

The sequence numbers of unacknowledged pdus transmitted on this connection can be ordered as:
[first-transmitted, last-transmitted + 1 (modulo 8)].
If N(R) is not in this interval, an error has occurred: nothing is acknowledged. If N(R) is equal to first-transmitted, nothing is acknowledged. If N(R) is in the interval and after "first-transmitted", then it implicitly acknowledges every I-pdu with a number between first-transmitted and N(R)-1 inclusive.

If the type byte indicates a response S-pdu and the F-bit is ONE, and there is an outstanding poll from this host, then N(R) implicitly acknowledges pdus as above, and any I-pdu which has been transmitted and not acknowledged is implicitly rejected. Note that the transport operation is full-duplex, so transmissions of the two parties can overlap. The failure of a received pdu to acknowledge a recent transmission may be an accident of timing, unless it is the explicit response to a "poll". When the receiver determines that an unacknowledged I-pdu has, in fact, been rejected, it must set its transmission sequence number (N(S)) back to the N(R) value appearing in the last remote acknowledgment, and retransmit all unacknowledged I-pdus on that connection.

If the type byte indicates a command pdu and P=1, the remote transport is demanding acknowledgment and the receiver must, at its earliest opportunity, send a supervisory response pdu with F=1 and an N(R) equal to the next expected sequence number as of receipt of the poll.

G.3.5. Use of Supervisory PDUs

There are two types of supervisory pdus: RR (receiver ready) and RNR (receiver not ready), either of which can be a command or a response.

Normally, when a host is required to acknowledge an I-pdu and has no data to transmit, it sends a RR response with F=0 and the appropriate N(R).

Normally, when a host is required to answer a poll, it sends an RR response with F=1 and the appropriate N(R).

When a host receives an in-sequence I-pdu and is unable to process it for want of buffers or whatever, or at any time a host determines that it does not want to receive I-pdus, it sends an RNR with N(R) appropriate to the last processed I-pdu and F=0. This identifies the host as "not ready". Once in this state, the transport acknowledges I-pdus with RNR responses with N(R) appropriate to the last accepted I-pdu. When answering a poll in this state, it sends an RNR response with F=1 and N(R) equal to the current expected sequence number. When a connection returns to the "ready" state, it sends a RR command with appropriate N(R) and P=0 or P=1 according to its desire to poll.

Accordingly, when a host receives a RNR, it should halt transmission of I-pdus, although it may still send supervisory pdus, until it receives a RR, indicating the clearing of the not-ready condition. Because of unpredictable timing, the presumption that any transmitted I-pdu which was not acknowledged by the RNR must be lost is not always true, and the practice is to poll on receipt of an RNR if there are unacknowledged I-pdus outstanding.

G.3.6. Timeouts

Whenever a transport in normal operation state issues a command with the P-bit set, it starts a timer. If the poll is not answered before the timer runs out, the transport issues a supervisory poll and reenters the polling state. The host counts successive polls and, if some reasonable maximum is exceeded, initiates a disconnect and enters the "disconnecting" state.

Note that waiting for an answer to a poll is independent of all other activity on the connection; the transport may be actively receiving and acknowledging pdus and still timeout the polling wait.

G.3.7. Deactivating the Connection

When the user requests a disconnect, or the transport initiates a disconnect because of some problem, the transport sends a DISC command pdu.

When a host receives a DISC, it answers it at its earliest opportunity with a UA response and cleans up its buffers, resets its counters, enters "disconnected mode" and presents a "disconnect indication" to the user.

When the transport which originates the DISC receives the UA response, it enters disconnected mode and presents a "disconnect confirmation" or "provider abort" to the user, depending on who initiated the disconnect, and enters the "disconnected" state. If the originator fails to receive a response to the DISC within the usual time limit, it repeats the DISC and restarts the timer.

If no response is received after some reasonable maximum number of retries, the originator proceeds as if the UA had been received. Once a DISC has been issued, no transmissions, except repeating the DISC or acknowledging a remote DISC command, are permissible.

G.3.8. Frame Formats

For this protocol, the pdu format is as follows:

Type, Segmentation, Text

where:

Type	is one byte designating the pdu type (see below);
Segment	is one byte conveying the segment number and end-of-message indicator;
Text	is a variable-length field of user information, appearing in information pdus only;

Format of the PDU Type byte:

Information (I) pdu:

```
 -----------------------------------
 | 0 |      N(S)     | P |   N(R)    |
 -----------------------------------
   7   6    5    4     3    2    1   0
```

```
where:
    bit 7      = 0, designates an I-pdu
    bits 4-6   = N(S) is the sequence number of this pdu
    bit 3      = P, the poll bit
    bits 0-2   = N(R) is the next sequence number expected
                 from the opposite host when this pdu
                 was transmitted (i.e.  N(S) from the last
                 received I-pdu plus 1)
```

Supervisory (S) pdu:

```
 --------------------------------
 | 1   0 | R | C |P/F|   N(R)   |
 --------------------------------
   7   6   5   4   3   2   1   0
```

```
where:
     bits 6-7 = 10 designates an S-pdu
     bit 5    = 0 for RR (receiver ready)
              = 1 for RNR (receiver not ready)
     bit 4    = 0 for response (bit 3 = F)
              = 1 for command  (bit 3 = P)
     bit 3    = P/F, the Poll/Final bit
     bits 0-2 = N(R), the next expected sequence number,
                as in the I-pdu above.
```

Unnumbered (U-)pdu:

```
 --------------------------------
 | 1   1 | M1  M2 |P/F| M3 M4 M5 |
 --------------------------------
   7   6   5   4   3   2   1   0
```

```
where:
    bits 6-7 = 11 designates a U-pdu
    bits 0-2 and 4-5 are the function code
    bit   3   is the Poll/Final bit; it is always
              ONE in SABM, DISC and UA, and always
              ZERO in DM.
```

values of the U-pdus,in hexadecimal, are:

```
    FC = SABM (set asynchronous balanced mode)
         command: initialize connection
    F0 = DM (disconnected mode)
         response: no active connection
    CA = DISC (disconnect)
         command: break connection

    CE = UA (unnumbered acknowledge)
         response: acknowledge SABM or DISC
```

Format of the Segment Byte:

```
    ---------------------------------
    | E |                           |
    | O |      Segment Number       |
    | M |                           |
    ---------------------------------
      7   6   5   4   3   2   1   0
```

where:

```
   bit 7 = EOM, the end-of-message (i.e. end of sdu)
           indicator;
   bits 0-6 = segment number, value 0 is the initial
           segment of an sdu.
```

Appendix H

Common-Memory Mapping Protocol

The common-memory mapping protocol is used by the application layer common-memory mapping service to communicate with a peer mapping service.

H.1 SERVICE DEFINITION

The function of the service is to copy local common-memory variables which are written by some local control or sensory or service process to any remote common-memory variables which map to these "original" variables.

The true mapping information is maintained by the network manager and distributed to the mapping service on each station as needed for implementation of that station's subset of the mappings. The network manager communicates to any given mapping service both the list of variables which it must transmit (and the associated receiving sites) and the list of variables which it will receive (and the associated transmitting sites). All of the logical connections are thus created by the network manager.

The operating connections are created by the individual mapping services on directive from the network manager. These connections are created through the transport connection service. As currently built, the AMRF network stations have only one application service, namely the memory mapping service, so no session or application selection is even performed.

As a consequence, the service has only two types of data:

a) outbound data, which is the contents of a local variable which is being copied to a remote memory; and

b) inbound data, which is the contents of a remote variable which is being copied to a local variable.

H.2 ELEMENTS OF THE PROTOCOL

This protocol has only one class of "protocol data unit" (pdu), called the "variable value".

The form of the variable-value pdu is:

Label, Sequence, Length, Value

where:

Label	is a two-byte identifier assigned to the "global data unit" represented by some pair of local and remote common-memory variables;

Sequence	is a two-byte value indicating in some way the "version" or "instance" of the variable value contained in this pdu. (This value is obtained from the local common-memory and merely copied by the service.)
Length	is a two-byte integer (most significant byte first) indicating the length in bytes of the value field of the pdu.
Value	is a variable length field comprising the (binary) value of the variable. The actual structure of this value depends on the "global data unit" being conveyed.

H.3 PROTOCOL SPECIFICATION

The mapping service maintains a list of "outbound" variables and a list of "inbound" variables according to directives of the network manager. Each list element identifies the local variable, the "global identifier" and the source or destination host.

When a variable in the "outbound" list changes value, as determined by a change in its "sequence" attribute, however locally implemented, the service constructs a variable-value pdu and presents it as a "data request" to the transport service for the connection associated with the destination host. The Label is the "global identifier" associated with the variable; the Sequence is the local sequence attribute value; the Length is the length in bytes of the significant text of the variable, however determined; and the Value is the literal binary value of that variable as locally stored.

It is not a requirement that every change to a local "outbound" variable be reflected in the remote variables. The rule is that the local service must transmit a variable-value pdu for this variable when the network service permits and only if the value has changed since it was last transmitted to that destination. This is a "best-effort" rule based on the AMRF common memory philosophy.

This rule permits the local service to minimize network overheads by utilizing the transport "data confirm" indication. Once a pdu for a given variable has been "transmitted" to its destination, no more pdus sending that variable to that destination will be constructed or transmitted until the preceding transmission is confirmed. This prevents a rapidly changing local variable from filling up network queues when there is a problem or a slow link.

When a mapping service receives a variable-value pdu, it attempts to locate a "global identifier" in its inbound list which matches the Label in the pdu. If no such match is found the pdu is discarded. Otherwise, the Length bytes of the Value field (or as many bytes as the local variable will accommodate if that is fewer) replace the current value of the matching local variable. The length and sequence attributes of the local variable are updated accordingly.

Appendix I

Source Code Listings

Listings for all common memory and network interface programs are available in hardcopy or computer readable form. Address your request, specifying which software and media, to:

AMRF Program Manager
National Bureau of Standards
Building 220, Room B111
Gaithersburg, MD 20899

GLOSSARY

activate	a term used interchangeably with "boot" to connote the act of starting the network services.
ATC	Turning Workstation Automated Turning Center controller
boot	a term used interchangeably with "activate" to connote the act of starting the network services.
CDWS	Cleaning and Deburring Workstation
CELL	cell controller
CM	an abbreviation for "common memory"
CMM	Coordinate Measuring Machine, a component of the Inspection Workstation.
common memory variable	a term used interchangeably with "mailbox" throughout the document.
DEMO	refers to the Sun Microsystems computer functioning as the front end common memory server identified in Figure II-6, operating under the (4.2 BSD) UNIX operating system.
HCSE	Hierarchical Control System Emulator
HGP	pedestal controller of the Horizontal Workstation. It works in cooperation with the Horizontal Workstation's Robot Control System (RCS, also known as HRC in this document).
HMB	material buffer controller, a component of the Horizontal Workstation. Also refered to as the Material Buffering Controller (MBC).
HMC	machine tool controller, a component of the Horizontal Workstation.
HRC	robot control system, a component of the Horizontal Workstation.
HVS	Vision System, support both the Horizontal Workstation, and the Material Handling Workstation.
HWSC	Horizontal Workstation Controller.

IMDAS	- the Integrated Manufacturing Data Administration System is the distributed data system which provides common interfaces to the AMRF's user programs and underlying databases.
IRC	- robot controller, a component of the Inspection Workstation.
IWS	- Inspection Workstation controller.
mail delivery table	- an data structure internal to the NIP. It contains the list of names for mailboxes that are to be received or transferred by the NIP.
mailbox	- logical storage area where messages (called "mailgrams") are placed by the sender process and picked up by one or more receiver processes.
mailgram	- a collection of contiguous bytes, stored in a mailbox.
MBC	- Material Buffering Controller (see also HMB, above).
MHS	- Material Handling System (also known as the MHWS - the Material Handling Workstation).
NIP	- Network Interface Process
pdu	- (see protocol data unit, below)
PPL	- Process Planning workstation
Praxis	- a strongly-typed structured language developed by Bolt, Beranek & Newman, Inc. A precursor to ADA.
protocol data unit	- (pdu) comes from the OSI model and means a string of bits which, taken together, form the basic unit of communication between peer services in a given layer of the model
sdu	- service data unit
SRI	- surface roughness instrument
Sun	- refers to a Sun Microsystems computer system. Within the AMRF network services, it refers primarily to the common memory front end system, called DEMO, although the network interfaces to CDWS, VWS, PPL, CELL, and MHS are all operating Sun systems.

Suntools menu	- refers to the list of options displayed on the Sun screen when the rightmost mouse button is pressed. Select an entry from that list by moving the mouse arrow to the desired option (it will be displayed in reverse video) and either releasing the mouse button OR clicking the leftmost mouse button while continuing to concurrently hold down the rightmost button
TWS	- Turning Workstation
variables	- when used in association with the phrase "common memory", it refers to common memory mailboxes.
VAX	- The VAX 11/785 supports the IMDAS and network manager functions of the AMRF. The operator interface assumes the use of the DEC VAX/VMS operating system.
VT100	- a descriptive term used to identify a general class of computer terminal equivalent in function and capability to the Digital Equipment Corporation VT100 terminal. Such a terminal is available from a large number of sources, including Qume, MicroTerm, etc.
VWS	- Vertical Workstation
Window	- refers to a delineated portion of the Sun video display screen that functions as a "terminal screen" for the application active within that delineated portion of the screen, much the same as the standard terminal screen.

REFERENCES

[1] Barbera, A. J., Fitzgerald, M. L., Albus, J. S., "Concepts for a Real-Time Sensory Interactive Control System Architecture", Procedings of the Fourteenth Southeastern Symposium on System Theory, April 1982, pp 121-126.

[2] Mitchell, M. and Barkmeyer, E., "Data Distribution in the NBS Automated Manufacturing Research Facility", Procedings of the National Symposium on Advances in Distributed Data Base Management for CAD/CAM, NASA Publication 2301, April 1984.

[3] Barbera, A. J., Fitzgerald, M. L., Albus, J. S., and Haynes, L. J., "RCS: The NBS Real-Time Robot Control System", Proceedings of the Robots VIII Converence, Detroit, Michigan, June 1984.

[4] "Advanced Data Communications Control Procedures", ANSI X3.66-1978, American National Standards Institute, New York, 1978.

[5] "Data Processing - Open Systems Interconnection - Basic Reference Model", ISO Standard 7498, International Standards Organization, Geneva, 1981.

[6] Carrier Sense Multiple Access with Collision Detection (CSMA/CD), IEEE Std 802.3-1985 (ISO/DIS 8802/3), The Institute of Electrical and Electronics Engineers, Inc, New York, 1984.

[7] Department of Defense, "Military Standard Transmission Control Protocol", MIL-STD-1778, August 1983

[8] Libes, D., "Experiences with a Communications Paradigm: Common Memory", in preparation.

[9] Logical Link Control, IEEE Std 802.2-1985 (ISO/DIS 8802/2), The Institute of Electrical and Electronics Engineers, Inc, New York, 1984.

[10] Holt, R. C., Graham, G. S., Lazowska, E. D., and Scott, M. A., Structured Concurrent Programming with Operating Systems Applications, Addison-Wesley Publishing Company, Reading, MA, 1978, p25.

[11] Furlani, C., Kent, E., Bloom, H., McLean, C., "The Automated Manufacturing Research Facility of the National Bureau of Standards", Proceedings of the Summer Simulation Conference, Vancouver, B.C., Canada, July 1983.

[12] Leffler, S., Fabry, R., Joy, W., "4.2BSD Interprocess Communications Primer", Computer Systems Research Group, U.C. Berkeley, 1983.

[13] Libes, D., "User-Level Shared Variables", Tenth USENIX Conference Proceedings, Summer 1985.

[14] Furlani, C.M., Editor, "Hierarchical Control System Emulation User's Manual", NBS-IR-85-3156, January 1985, 130 pp.

[15] Furlani, C.M., Editor, "Hierarchical Control System Emulation Programmer's Manual", NBS-IR-85-3157, January 1985, 45 pp.

[16] Johnson, T.L., Milligan, S.D., Fortmann, T.E., "Hierarchical Control System Emulation Applications Guide", NBS-GCR-82-410, October 1982, 115 pp.

[17] Electronic Industries Association, EIA Standard RS-232-C, Washington, D.C., 1969.

[18] Token-Passing Bus Access Method and Physical Layer Specifications, IEEE Std 802.4-1985 (ISO/DIS 8802/4), The Institute of Electrical and Electronics Engineers, Inc, New York, 1984.

[19] Electronic Industries Association, EIA Standard RS-449 Washinton, D.C.

[20] Department of Defense, "Military Standard Internet Protocol", MIL-STD-1777, August 1983.

[21] International Organization for Standardization, "Connection Oriented Transport Protocol", DP 8073, 1983

[22] Fletcher, J. "An Arithmetic Checksum for Serial Transmissions", IEEE Transactions on Communications, January 1982.

[23] Libes, D., "Byte Ordering", in preparation.

[24] International Organization for Standardization, "Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)", ISO 8824/8825, 1984

[25] Libes, D., Barkmeyer, E., "The Integrated Manufacturing Data Administration System (IMDAS) -- an overview", Int. J. Computer Integrated Manufacturing, Vol. 1, No. 1, pp. 44-49.

[26] Furlani et al, "The Integrated Manufacturing Data Administration System (IMDAS)", to be published as an NBSIR, 1988.

READER COMMENT FORM

Document Title: AMRF Network Communications

This document is one in a series of publications which document research done at the National Bureau of Standards' Automated Manufacturing Research Facility from 1981 through March, 1987.

You may use this form to comment on the technical content or organization of this document or to contribute suggested editorial changes.

Comments: ______________________________

____

____

____

____

____

____

____

If you wish a reply, give your name, company, and complete

mailing address: ______________________________

____

____

____

What is your occupation? ______________________________

NOTE: This form may not be used to order additional copies of this document or other documents in the series. Copies of AMRF documents are available from NTIS.

Please mail your comments to: AMRF Program Manager
National Bureau of Standards
Building 220, Room B-111
Gaithersburg, MD 20899

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBSIR 88-3816	2. Performing Organ. Report No.	3. Publication Date JUNE 1988

4. TITLE AND SUBTITLE

AMRF Network Communications

5. AUTHOR(S)

Rybczynski, S., Barkmeyer, E.J., Wallace, E.K, Strawbridge, M.L., Libes, D.E., Young, C.V

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

This document discusses the 1986 version of the factory data communications component of the National Bureau of Standards' Automated Manufacturing Research Facility. The underlying architecture, protocols, hardware, software and manual procedures are detailed.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

AMRF, networking, communications, common memory, shared memory

13. AVAILABILITY

[X] Unlimited
☐ For Official Distribution. Do Not Release to NTIS
☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
[X] Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

206

15. Price

$24.95